Содержание

ВВЕДЕНИЕ

В информатике под формальным языком понимается язык программирования, построенный по правилам некоторого логического исчисления и представляющий собой систему построения конечных знаков последовательностей в заданном алфавите. Теория формальных языков является разделом математической лингвистики – науки, изучающей языки и формальные методы их построения. Теория формальных языков включает в себя способы описания формальных грамматик языков, построение методов и алгоритмов анализа принадлежности цепочек языку, а также алгоритмов перевода (трансляции) алгоритмических языков на язык машины. Теория формальных языков зародилась в конце 50-х годов XX века, когда команда под руководством американского ученого Джона Бэкуса разработала первый высокоуровневый язык программирования Фортран. В своей деятельности команда Бэкуса основывалась на научных разработках лингвиста Ноама Хомского, создавшего классификацию формальных языков.

Разработка нового языка программирования требует творческого подхода и кропотливой работы, несмотря на существование большого количества алгоритмов для автоматизации процесса написания транслятора для формальных языков. Это касается синтаксиса языка, который должен быть как удобен в прикладном программировании, так и должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа.

Целью данной курсовой работы является разработка распознавателя модельного языка программирования, согласно заданной формальной грамматике, основанная на индивидуальном варианте. Для достижения цели необходимо выполнить следующие задачи:

- освоение основных методов разработки распознавателей формальных языков на примере модельного языка программирования;

- приобретение практических навыков по написанию транслятора языка программирования;
- закрепление практических навыков самостоятельного решения инженерных задач, умения пользоваться справочной литературой и технической документацией.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Постановка задачи

Разработать распознаватель модельного языка программирования, согласно заданной формальной грамматике на основе индивидуального варианта.

Распознаватель представляет собой алгоритм, позволяющий вынести решение о принадлежности цепочки символов некоторому языку.

Распознаватель схематично представляется в виде совокупности входной ленты, читающей головки, указывающей на очередной символ на ленте, устройства управления (далее сокращение УУ) и дополнительной памяти.

Конфигурацией распознавателя является:

- состояние УУ;
- содержимое входной ленты;
- положение читающей головки;
- содержимое дополнительной памяти.

Трансляция исходного текста программы осуществляется в несколько этапов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация целевого кода.

Лексический анализ является наиболее простой фазой и выполняется с помощью регулярной грамматики. Регулярным грамматикам соответствуют конечные автоматы, следовательно, разработка и написание программы лексического анализатора эквивалентна разработке конечного автомата и его диаграммы состояний (далее сокращение ДС).

Алгоритм синтаксического анализа строится на базе контекстно-свободных (далее сокращение КС) грамматик. Задача синтаксического анализатора − провести анализ разбор текста программы и сопоставить его с формальным описанием языка.

Семантический анализ позволяет учесть особенности языка, которые не могут быть описаны правилами КС-грамматики. К таким особенностям относятся:

- обработка описаний;
- анализ выражений;
- проверка правильности операторов.

Обработка описаний позволяет убедиться в том, что каждая переменная в программе описана и только один раз.

Анализ выражений заключается в том, чтобы проверить описаны ли переменные, участвующие в выражении, и соответствуют ли типы операндов друг другу и типу операции.

Этапы синтаксического и семантического анализа обычно объединяют.

1.2 Порядок выполнения

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики, вариант № 6.
2. Составить таблицу лексем и нарисовать диаграмму состояний для распознавания и формирования лексем языка.

3. Разработать процедуру лексического анализа исходного текста программы на языке высокого уровня.
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке высокого уровня.
5. Построить программный продукт, читающий текст программы, написанной на модельном языке, в виде консольного приложения.
6. Протестировать работу программного продукта с помощи серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки.

1.3 Грамматика языка

Согласно индивидуальному варианту № 6 курсовой работы, грамматика языка включает следующие синтаксические конструкции:

1. <операции_группы_отношения>:: = NE | EQ | LT | LE | GT | GE;
2. <операции_группы_сложения>:: = + | -| or;
3. <операции_группы_умножения>::= * | / | and;
4. <унарная_операция>::= not
5. <программа>::= «{» {/ (<описание> | <оператор>) ; /} «}»
6. <описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}
7. <тип>::= integer | real | boolean
8. <составной>::= begin <оператор> { ; <оператор> } end
9. <присваивания>::= <идентификатор> := <выражение>
10. <условный>::= if «(»<выражение> «)» <оператор> [else <оператор>]
11. <фиксированного_цикла>::= for <присваивания> to <выражение> do <оператор>;
12. <условного_цикла>::= while <выражение> do <оператор>;
13. <ввода>::= read «(»<идентификатор> {, <идентификатор> } «)»;

14. <вывода>::= write «(»<выражение> {, <выражение> } «)»;
15. <выражение>::= <операнд> {<операции_группы_отношения> <операнд>};
16. <операнд>::= <слагаемое> {<операции_группы_сложения> <слагаемое>};
17. <слагаемое>::= <множитель> {<операции_группы_умножения> <множитель>};
18. <множитель>::= <идентификатор> | <число> | <логическая_константа> | <унарная_операция> <множитель> | «(» <выражение> «)»;
19. <логическая_константа> ::= true | false;
20. <идентификатор>::= <буква> {<буква> | <цифра>};
21. <число>::= <цифра> {<цифра>};
22. <буква>::= 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 | 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;
23. <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Здесь для записи правил грамматики используется форма Бэкуса-Наура (БНФ). В записи БНФ левая и правая части порождения разделяются символом «::=», нетерминалы заключены в угловые скобки, а терминалы − просто символы, используемые в языке. Терминалы, представляющие собой ключевые слова языка:

- <>;
- =;
- <;
- <=;
- >;
- >=;
- or;

- +;
- -;
- *;
- /;
- integer;
- real;
- boolean;
- as;
- if;
- then;
- else;
- for;
- to;
- do;
- while;
- read;
- write;
- not;
- true;
- false;

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Разработка лексического анализатора

Лексический анализатор – это подпрограмма, принимающая на вход исходный код программы и выдающая последовательность лексем – минимальных элементов программы, несущих смысловую нагрузку. Исходный код программы записан в текстовом файле Code.txt.

В модельном языке программирования были выделены следующие типы лексем:

- ключевые слова;
- ограничители;
- числа;
- идентификаторы.

Во время разработки лексического анализатора ключевые слова и разделители выделяются заранее в отдельных файлах KeyWords.txt и Delimeters.txt соответственно, идентификаторы и числа в момент разбора исходного кода и записываются в Identificators.txt и Numbers.txt.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы − пара чисел n,k, где n − номер таблицы, а k − номер лексемы в таблице. Данные записываются в Lexems.txt.

Кроме того, в исходном коде программы кроме ключевых слов, идентификаторов и числовых констант может находиться произвольное число пробельных символов («пробел», «табуляция», «возврат каретки»), комментариев, завершение строки с помощью символа «точка с запятой» и комментарии, заключенные в фигурные скобки.

Лексический анализ текста проводится по регулярной грамматике.

Известно, что регулярная грамматика эквивалентна конченому автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата. Диаграмма состояний представлена на Рисунке 1.

Исходный код лексического анализатора приведен в Приложении А.

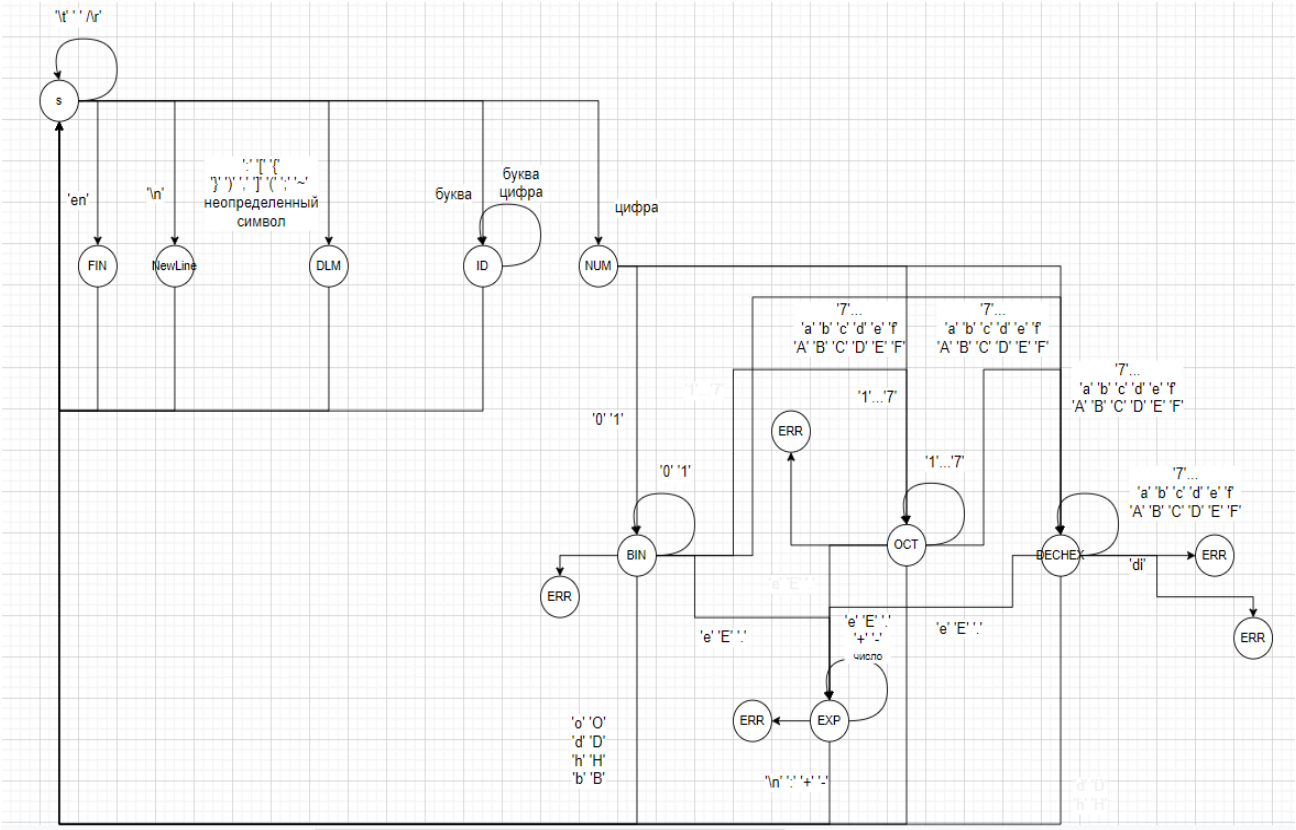

Рисунок 1 – **Диаграмма состояний**

2.2 Разработка синтаксического анализатора

Предположим, что лексический и синтаксический анализаторы взаимодействуют следующим образом: если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора.

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (далее сокращение РС). В основе метода лежит тот факт, что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

- P → program var D1|B begin {; | \n D1|B} end. ;
- D1 → D {,D};
- D → I {,I}[integer | real | boolean];

- B → S ;
- S → I as *E*| if E then S [else S | **ε**] | for I as E to E do B | while E do S | B | read(I{,I}) | write(E{,E}) | [S {; | \n S}];
- E → E1{[<> | = | < | <= | > | >=] E1};
- E1 → T{[+ | − | or] T};
- T → F{[∗ | / | and] F};
- F → I | N | L| ~ F | (E);
- L → true | false;
- I → C | IC | IR;
- N → R | NR;
- C → a | b | … | z | A | B | … | Z; - R → 0 | 1 | … | 9.

Здесь правила для нетерминалов L, I, N, C и R описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов P, D1, D, B, S, E, E1, T, F.

Исходный код синтаксического анализатора приведен в Приложении А.

2.3 Семантический анализ

К сожалению. некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. К таким правилам относятся:

- любой идентификатор, используемый в теле программы должен быть описан;
- повторное описание одного и того же идентификатора не разрешается.

Указанные особенности языка разбираются на этапе семантического анализа. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. На практике это означает, что в рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки. Например, на этапе лексического анализа в таблицу TID заносятся данные обо всех лексемах-идентификаторах, которые встречаются в тексте программы. На этапе

синтаксического анализа при получении каждого идентификатора информация о нем заносится в буферную память, если он еще не объявлен. В случае использования идентификатора, отсутствующего в буферной памяти, ошибка об этом выводится пользователю. В случае повторной инициализации идентификатора, уже присутствующего в буферной памяти, ошибка об этом также выводится пользователю.

Описания функций семантических проверок приведены в листинге в Приложении Б.

3 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В качестве программного продукта разработано консольное приложение. Приложение принимает на вход исходный текст программы на модельном языке и выдает в качестве результата сообщение о синтаксической и семантической корректности написанной программы. В случае обнаружения ошибки программа выдает сообщение об ошибке с номером некорректной лексемы. Список ошибок представлен в Таблице 1.

Таблица 1 − Список ошибок

Номер ошибки	**Суть ошибки**
101	Ожидалось число
102	Ожидалось }
103	Ожидалось число или e или E или. или b или B
104	Ожидалось число или e или E или. или o или O

Продолжение Таблицы 1

105	Ожидалось отсутствие точки после e
106	Ожидалось e или E
107	Ожидалось отсутствие i после d
108	Ожидалось число или e или E или. или d или D или h или H
109	Ожидалось отсутствие < после >
110	Ожидался разделитель
111	Ожидалась буква для описания cc
112	Ожидалось продолжение действительного числа
113	Ожидалось продолжение действительного числа
114	Ожидалось D или d или B или b или H или h или O или o или . или E или e или число
201	Ожидался оператор группы умножения
201	Ожидался оператор группы умножения
202	Ожидался оператор или описание
203	Ожидалось ; или перенос строки
204	Ожидалось }
205	Ожидался идентификатор
206	Неопознанная ошибка
207	Ожидался тип переменной
208	Ожидался]
209	Ожидалось ключевое слово as
210	Ожидалось ключевое слово then
211	Ожидалось ключевое слово to
212	Ожидалось выражение
213	Ожидалось ключевое слово do
214	Ожидался оператор
215	Ожидалась (
216	Ожидалась операция группы сложения
217	Ожидалась)

Продолжение Таблицы 1

218	Неопознанная ошибка
219	Ожидалась операция группы отношения
221	Ожидалась унарная операция not
301	Повторное объявление переменной
302	Использование необъявленной переменной

Рассмотрим примеры.

3.1 Тест 1. Правильный код

Листинг 1 – Код теста 1

```
{
a , b , c , d , d as integer ;
k as real ;
r , f as boolean ;
r as 3D ;
write ( a ) ;
}
```

Программа выдаст информацию об успешном прохождении теста. Результат работы программы при тесте 1 представлен на Рисунке 2.

Рисунок 2 – Результат работы программы при тесте 1

3.2 Тест 2. Лексическая ошибка

Для проверки правильности программы, во 2 тесте была допущена умышленная лексическая ошибка. В 5 строке кода был добавлен символ «$» не обусловленный правилами грамматики.

Листинг 2 – Код теста 2

```
{
integer a, b, c, d
$
read(a);
b as 001b;
c as 5d plus 7h;
write(d);
}
```

Программа должна выдать ошибку при прохождении лексического анализа. Так как последнее, на что проверяет лексический анализатор, это разделители, то в случае успешного тестирования, программа должна выдать ошибку 110, что согласно Таблице 1 характеризуется как «Ожидался разделитель.». Результат работы программы при тесте 2 представлен на Рисунке 3.

Рисунок 3 – Результат работы программы при тесте 2

3.3 Тест 3. Синтаксическая ошибка

Для проверки правильности программы, в 3 тесте была допущена умышленная синтаксическая ошибка. В 4 строке кода было убрано ключевое слово «as», что нарушает правила грамматики.

Листинг 3 – Код теста 3

```
{
a, b, c, d integer
read(a);
b  001b;
c as 5d plus 7h;
write(d);
}
```

Программа должна выдать ошибку при прохождении синтаксического анализа. В случае успешного тестирования, программа должна выдать ошибку 209, что согласно Таблице 1 характеризуется как «ожидалось ключевое слово as». Результат работы программы при тесте 3 представлен на Рисунке 4.

Рисунок 4 – Результат работы программы при тесте 3

3.4 Тест 4. Семантическая ошибка

Для проверки правильности программы, в 4 тесте была допущена умышленная семантическая ошибка. Во 5 строке кода была добавлена необъявленная переменная «t», что нарушает семантические условия.

Листинг 4 – Код теста 4

```
{
a, b, c, d, d as integer ;
e as real ;
r, f as boolean ;
t as 3D ;
}
```

Программа должна выдать ошибку при прохождении синтаксического анализа на этапе семантической проверки переменных. В случае успешного

тестирования, программа должна выдать ошибку 302, что согласно Таблице 1 характеризуется как «Использование необъявленной переменной». Результат работы программы при тесте 4 представлен на Рисунке 5.

Рисунок 5 – Результат работы программы при тесте 4

Таким образом, после прохождения всех приведённых проверок можно сказать, что программа работает корректно.

ЗАКЛЮЧЕНИЕ

В ходе курсового проектирования был разработан распознаватель модельного языка, который включает в себя: лексический, синтаксический и семантический анализаторы. Лексический анализатор, разделяющий последовательность символов исходного текста программы на последовательность лексем. Лексический анализатор реализован на языке высокого уровня Python.

Разбор исходного текста программы был сделан с помощью синтаксического анализатора, который реализован также на языке Python. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости была преобразована грамматика, в частности, специальным образом обработаны встречающиеся итеративные синтаксически конструкции (нетерминалы D, D1, B, E1 и T).

В код рекурсивных функций включены проверки семантических условий − проверка на повторное объявление одной и той же переменной и проверка на использование необъявленной переменной.

Тестирование приложения показало, лексический, синтаксический и семантический анализаторы работают корректно, написанная программа успешно распознается анализатором, а программа, содержащая ошибки, выдает ошибки с кратким описанием сути ошибки.

В ходе работы изучены основные принципы построения систем на основе теории автоматов и формальных грамматик, приобретены навыки лексического, синтаксического и семантического анализа предложений языков программирования. Были закреплены и изучены знания языка программирования Python. Так же, для упрощения работы, были импортированы сторонние Python библиотеки: numpy и isHex. Программа курсовой работы была разделена на 5 Python файлов, это было сделано для того, чтобы можно было

незатруднительно работать с модулями и по отдельности реализовывать анализаторы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Свердлов С.З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019.

2. Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2020.

3. Миронов С.В. Формальные языки и грамматики: учебное пособие для студентов факультета компьютерных наук и информационных технологий. – Саратов: СГУ, 2019.

4. Антик М.И., Казанцева Л.В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.

5. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.

ПРИЛОЖЕНИЯ

Приложение А – Код лексического анализатора

Приложение Б – Код синтаксического анализатора (с семантическими проверками)

Приложение А

Код лексического анализатора

```
import GenerateFiles
import main
from isHex import isHex
import numpy as np

"""
1 - KEYWORDS
2 - DELIMETERS
3 - IDENTIFICATORS
4 - NUMBERS
"""


def Lexer():
    # ЕСЛИ ЭТО ID
    IsID = True

    # БУФЕР ДЛЯ СЛОВА
    BufWord = ""

    # БУФЕР ДЛЯ ЧИСЕЛ
    BufNum = ""

    # ДВУМЕРНЫЙ МАССИВ ЛЕКСЕММ
    Lexems = np.empty((512, 2), dtype="object")

    # СТРИНГОВЫЙ МАССИВ, КУДА БУДЕМ ЗАПИСЫВАТЬ КЛЮЧЕВЫЕ СЛОВА
    KeyWordRead = []

    # СТРИНГОВЫЙ МАССИВ, КУДА БУДЕМ ЗАПИСЫВАТЬ КЛЮЧЕВЫЕ РАЗДЕЛИТЕЛИ
    DelimetersRead = []

    CurrentLexem = 0  #

    # НЫНЕШНЕЕ СОСТОЯНИЕ АВТОМАТА
    CurrentCondition = "H"

    # В WORDCHAR ЗАПИСЫВАЕМ НАШ ФАЙЛ
```

```
main.WordChar = main.Word.read()
# ПЕРЕОПРЕДЕЛЯЕМ МАССИВ В ЛИСТ, ЧТОБЫ КАЖДЫЙ CHAR БЫЛ ОТДЕЛЬНО
main.WordChar = list(main.WordChar)

# print(main.WordChar) DEBUG

# В FileKeyWords ЗАПИСЫВАЕМ ДАННЫЕ ФАЙЛА
FileKeyWords = open("KeyWords.txt", 'r')

# СЧИТЫВАЕМ СТРОКИ
TempKeyWordRead = FileKeyWords.readlines()

# ЦИКЛ WHILE ПРЕДНАЗНАЧЕН ДЛЯ ПЕРЕНОСА ЗНАЧЕНИЙ ИЗ TempKeyWordRead
# В KeyWordRead БЕЗ \n
count = 0
LenTKWR = len(TempKeyWordRead) - 1
while LenTKWR >= count:
    TempChar = TempKeyWordRead[count]
    TempChar = TempChar[0:len(TempKeyWordRead[count]) - 3]
    if TempChar[-1] == ' ':
        TempChar = TempChar[:-1]
    KeyWordRead.append(TempChar)
    count += 1

# print(KeyWordRead) # DEBUG

main.KeyWord = KeyWordRead

print(main.KeyWord)

# В FileDelimetersRead ЗАПИСЫВАЕМ ДАННЫЕ ФАЙЛА
FileDelimetersRead = open("Delimeters.txt", 'r')

# СЧИТЫВАЕМ СТРОКИ
TempDelimetersRead = FileDelimetersRead.readlines()

# ЦИКЛ WHILE ПРЕДНАЗНАЧЕН ДЛЯ ПЕРЕНОСА ЗНАЧЕНИЙ ИЗ TempDelimetersRead
# В DelimetersRead БЕЗ \n
count = 0
LenTDR = len(TempDelimetersRead) - 1
while LenTDR >= count:
```

```
        TempChar = TempDelimetersRead[count]
        TempChar = TempChar[0:len(TempDelimetersRead[count]) - 3]
        if TempChar[-1] == ' ':
            TempChar = TempChar[:-1]
        DelimetersRead.append(TempChar)
        count += 1

    DelimetersRead[8] = '\n'
    # print(DelimetersRead)  # DEBUG

    main.Delimeters = DelimetersRead

    print(main.Delimeters)

    # ПОКА ПОЗИЦИЯ АВТОМАТА МЕНЬШЕ ДЛИНЫ МАССИВА, СОСТОЯЩЕГО ИЗ ЧАРОВ
    # CurrentCondition - СОСТОЯНИЕ АВТОМАТА
    # CurPosition - СОСТОЯНИЕ АВТОМАТА
    LenMainChar = len(main.WordChar)
    while main.CurPosition < LenMainChar:

        # ЕСЛИ СОСТОЯНИЕ АВТОМАТА РАВНО 'H'
        if CurrentCondition == "H":

            if main.WordChar[main.CurPosition] == '\t' or
main.WordChar[main.CurPosition] == ' ' or main.WordChar[
                main.CurPosition] == '\r':
                main.CurPosition += 1

            elif main.WordChar[main.CurPosition] == '}':
                # FIN
                CurrentCondition = "FINISH"

            elif main.WordChar[main.CurPosition] == '\n':
                # NewLine
                CurrentCondition = "NEWLINE"

            # ДОБАВИЛ ; надо проверить
            elif main.WordChar[main.CurPosition] == '{' or
main.WordChar[main.CurPosition] == ':' or main.WordChar[
                main.CurPosition] == '[' or main.WordChar[main.CurPosition] ==
'/' and main.WordChar[
```

```
                main.CurPosition + 1] == '*' or
main.WordChar[main.CurPosition] == '*' and main.WordChar[
                main.CurPosition + 1] == '/' or
main.WordChar[main.CurPosition] == ')' or main.WordChar[
                main.CurPosition] == ',' or main.WordChar[main.CurPosition] ==
']' or main.WordChar[
                main.CurPosition] == '(' or main.WordChar[main.CurPosition] ==
'not' or main.WordChar[
                main.CurPosition] == ';':
                main.Sost = False
                # DLM
                CurrentCondition = "DELIMETER"

            elif main.WordChar[main.CurPosition].isalpha():
                BufWord = ""
                # ID
                CurrentCondition = "ID"

            elif main.WordChar[main.CurPosition].isdigit():
                main.IsLetter = False
                BufNum = ""
                # NM
                CurrentCondition = "NUMBER"

            else:
                main.Sost = False
                # DLM
                CurrentCondition = "DELIMETER"

        elif CurrentCondition == "NEWLINE":

            # Lexems[CurrentLexem][0] = str(2)  # ВТОРАЯ ТАБЛИЦА - DELIMETERS
            # Lexems[CurrentLexem][1] = str(9)  # ДЕВЯТАЯ СТРОЧКА В ТАБЛИЦЕ
DELIMETERS

            Lexems[CurrentLexem, 0] = str(2) # +
            Lexems[CurrentLexem, 1] = str(9)  # +

            CurrentLexem += 1
            main.CurPosition += 1
```

```
            # ВОЗВРАЩАЕМСЯ В НАЧАЛЬНОЕ СОСТОЯНИЕ АВТОМАТА
            CurrentCondition = "H"

        elif CurrentCondition == "FINISH":

            # print("main.WordChar: ", main.WordChar)
            # print("len(main.WordChar): ", len(main.WordChar))
            # print("main.WordChar[main.CurPosition + 1]",
main.WordChar[main.CurPosition + 1])
            # print("main.WordChar[main.CurPosition + 2]",
main.WordChar[main.CurPosition + 2])
            # print("main.WordChar[main.CurPosition + 3]",
main.WordChar[main.CurPosition + 3])

            if main.WordChar[main.CurPosition] == '}':

                # Lexems[CurrentLexem][0] = str(1)  # ПЕРВАЯ ТАБЛИЦА -
KEYWORDS
                # Lexems[CurrentLexem][1] = str(16)  # ШЕСТНАДЦАТАЯ СТРОЧКА В
ТАБЛИЦЕ KEYWORDS

                Lexems[CurrentLexem, 0] = str(2)
                Lexems[CurrentLexem, 1] = str(4)

                CurrentLexem += 1
                main.CurPosition = main.CurPosition + 1
                CurrentCondition = "H"  # ВОЗВРАЩАЕМСЯ В НАЧАЛЬНОЕ СОСТОЯНИЕ
АВТОМАТА
                break

            else:
                ErrorLexer(102)  # ПЕРЕДАЁМ ОШИБКУ



        # ID
        elif CurrentCondition == "ID":

            if main.WordChar[main.CurPosition].isdigit() or
main.WordChar[main.CurPosition].isalpha():
                BufWord += main.WordChar[main.CurPosition]
```

```
        # print(BufWord)
        # print("148: ", GetLexems(BufWord)[0])

        if GetLexems(BufWord)[0] != 0:
            main.CurPosition += 1
            CurrentCondition = "ID"

            # Lexems[CurrentLexem][0] = str(GetLexems(BufWord)[0])
            # Lexems[CurrentLexem][1] = str(GetLexems(BufWord)[1])

            Lexems[CurrentLexem, 0] = str(GetLexems(BufWord)[0])
            Lexems[CurrentLexem, 1] = str(GetLexems(BufWord)[1])

            # print( Lexems[CurrentLexem,0])
            # print( Lexems[CurrentLexem,1])
            # exit(0)

            IsID = True

        else:
            IsID = False

            # Lexems[CurrentLexem][0] = str(3)
            # Lexems[CurrentLexem][1] = str(main.CurID + 1)

            Lexems[CurrentLexem, 0] = str(3)
            Lexems[CurrentLexem, 1] = str(main.CurID + 1)

            main.CurPosition += 1
            CurrentCondition = "ID"
    else:
        if not (IsID):
            # print("BufWord: ", BufWord)
            # print(main.CurID)

            # main.Identificators[main.CurID] = BufWord

            main.Identificators.append(BufWord)
            main.CurID += 1
        CurrentLexem += 1
```

```
                CurrentCondition = "H"

        elif CurrentCondition == "NUMBER":

            if main.WordChar[main.CurPosition] == '0' or
main.WordChar[main.CurPosition] == '1':
                # BIN
                CurrentCondition = "BIN"

            elif int(main.WordChar[main.CurPosition]) > 1 and
int(main.WordChar[main.CurPosition]) < 8:
                # OCT
                CurrentCondition = "OCT"

            elif int(main.WordChar[main.CurPosition]) > 7 or
isHex(main.WordChar[main.CurPosition]):
                CurrentCondition = "DECHEX"

            else:
                ErrorLexer(101)  # ОШИБКА

        elif CurrentCondition == "BIN":

            BufNum  +=  main.WordChar[main.CurPosition]

            if main.WordChar[main.CurPosition] == 'O' or
main.WordChar[main.CurPosition] == 'o':
                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                NumberOfSystem = "OCT"
                main.Numbers[main.CurNum] = GetNum(BufNum, NumberOfSystem)
                CurrentLexem += 1
                main.CurNum += 1
```

```
            elif main.WordChar[main.CurPosition] == 'd' or
main.WordChar[main.CurPosition] == 'D':

                if main.WordChar[main.CurPosition] == 'd' and
main.WordChar[main.CurPosition] == 'i':
                    ErrorLexer(107)

                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                main.Numbers[main.CurNum] = BufNum[0:len(BufNum) - 1]
                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'H' or
main.WordChar[main.CurPosition] == 'h':
                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                NumberOfSystem = "HEX"
                main.Numbers[main.CurNum] = GetNum(BufNum, NumberOfSystem)
                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'B' or
main.WordChar[main.CurPosition] == 'b':
                main.IsLetter = True
                CurrentCondition = "H"
```

```
                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                NumberOfSystem = "BIN"

                # main.Numbers[main.CurNum] = GetNum(BufNum, NumberOfSystem)

                main.Numbers.append(GetNum(BufNum, NumberOfSystem))
                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'e' or
main.WordChar[main.CurPosition] == 'E' or main.WordChar[
                main.CurPosition] == '.':
                main.CurPosition -= 1
                BufNum = BufNum[0:len(BufNum) - 1]
                CurrentCondition = "EXP"

            elif int(main.WordChar[main.CurPosition]) > 1 and
int(main.WordChar[main.CurPosition]) < 8:
                CurrentCondition = "OCT"

            elif main.WordChar[main.CurPosition] == '0' or
main.WordChar[main.CurPosition] == '1':
                CurrentCondition = "BIN"

            elif int(main.WordChar[main.CurPosition]) > 7 or
isHex(main.WordChar[main.CurPosition]):
                CurrentCondition = "DECHEX"

            else:
                ErrorLexer(103)

            main.CurPosition += 1

        elif CurrentCondition == "OCT":

            BufNum += main.WordChar[main.CurPosition]
```

```
            if main.WordChar[main.CurPosition] == 'O' or
main.WordChar[main.CurPosition] == 'o':
                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)  # ЧЕТВЁРТАЯ ТАБЛИЦА -
IDENTIFICATORS
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)
                NumberOfSystem = "OCT"

                # main.Numbers[main.CurNum] = GetNum(BufNum, NumberOfSystem)

                main.Numbers.append(GetNum(BufNum, NumberOfSystem))

                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'd' or
main.WordChar[main.CurPosition] == 'D':

                if main.WordChar[main.CurPosition] == 'd' and
main.WordChar[main.CurPosition] == 'i':
                    ErrorLexer(107)

                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                # main.Numbers[main.CurNum] = BufNum[0:len(BufNum) - 1]

                main.Numbers.append(BufNum[0:len(BufNum) - 1])
```

```
                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'H' or
main.WordChar[main.CurPosition] == 'h':
                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                NumberOfSystem = "HEX"

                # main.Numbers[main.CurNum] = GetNum(BufNum, NumberOfSystem)

                main.Numbers.append(GetNum(BufNum, NumberOfSystem))

                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'e' or
main.WordChar[main.CurPosition] == 'E' or main.WordChar[
                main.CurPosition] == '.':
                main.CurPosition -= 1
                BufNum = BufNum[0:len(BufNum) - 1]
                CurrentCondition = "EXP"

            elif int(main.WordChar[main.CurPosition]) > 1 and
int(main.WordChar[main.CurPosition]) < 8:
                CurrentCondition = "OCT"

            elif main.WordChar[main.CurPosition] == '0' or
main.WordChar[main.CurPosition] == '1':
                CurrentCondition = "BIN"

            elif int(main.WordChar[main.CurPosition]) > 7 or
isHex(main.WordChar[main.CurPosition]):
                CurrentCondition = "DECHEX"
```

```
            else:
                ErrorLexer(104)

            main.CurPosition += 1
        # ФУЛЛ ИЗУЧИТЬ И ПЕРЕПРОВЕРИТЬ
        elif CurrentCondition == "EXP":
            if main.WordChar[main.CurPosition] == 'd' or
main.WordChar[main.CurPosition] == 'D' or main.WordChar[
                main.CurPosition] == 'h' or main.WordChar[main.CurPosition] ==
'h' or main.WordChar[
                main.CurPosition] == 'o' or main.WordChar[main.CurPosition] ==
'O' or main.WordChar[
                main.CurPosition] == 'b' or main.WordChar[main.CurPosition] ==
'B':
                ErrorLexer(112)

            if main.WordChar[main.CurPosition] == 'e' or
main.WordChar[main.CurPosition] == 'E':
                main.IsLetter = True
                BufNum += main.WordChar[main.CurPosition]
                main.Exponent = True

            elif main.WordChar[main.CurPosition] == '.' and main.Exponent ==
True:
                ErrorLexer(105)

            elif main.WordChar[main.CurPosition] == '.' and main.Exponent ==
False:
                BufNum += main.WordChar[main.CurPosition]

            elif main.WordChar[main.CurPosition].isdigit():
                BufNum += main.WordChar[main.CurPosition]

            elif (main.WordChar[main.CurPosition] == '+' or main.WordChar[
                main.CurPosition] == '-') and main.Exponent == False:
                ErrorLexer(106)

            elif main.WordChar[main.CurPosition] == '\n' or
main.WordChar[main.CurPosition] == ';':
                # ЧЕТВЁРТАЯ ТАБЛИЦА - IDENTIFICATORS
```

```
                Lexems[CurrentLexem][0] = str(4)
                Lexems[CurrentLexem][1] = str(main.CurNum + 1)
                print(BufNum[0:len(BufNum)])
                # print("5.e+1")
                main.Numbers.append(BufNum[0:len(BufNum)])
                CurrentLexem += 1
                main.CurNum += 1

                if main.WordChar[main.CurPosition] == '\n':
                    # Lexems[CurrentLexem][0] = str(2)
                    # Lexems[CurrentLexem][1] = str(9)

                    Lexems[CurrentLexem, 0] = str(2)
                    Lexems[CurrentLexem, 1] = str(9)

                    CurrentLexem += 1
                CurrentCondition = "H"

            elif main.WordChar[main.CurPosition].isalpha() and (
                    main.WordChar[main.CurPosition] != 'e' or
main.WordChar[main.CurPosition] != 'E'):
                ErrorLexer(113)
            else:
                BufNum += main.WordChar[main.CurPosition]
            main.CurPosition += 1

        elif CurrentCondition == "DECHEX":

            BufNum += main.WordChar[main.CurPosition]

            if main.WordChar[main.CurPosition] == 'H' or
main.WordChar[main.CurPosition] == 'h':
                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)  # ЧЕТВЁРТАЯ ТАБЛИЦА -
IDENTIFICATORS
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)
```

```
                main.Numbers[main.CurNum] = BufNum[0:len(BufNum) - 1]
                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'd' or
main.WordChar[main.CurPosition] == 'D':

                if main.WordChar[main.CurPosition] == 'd' and
main.WordChar[main.CurPosition] == 'i':
                    ErrorLexer(107)

                main.IsLetter = True
                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] = str(4)
                # Lexems[CurrentLexem][1] = str(main.CurNum + 1)

                Lexems[CurrentLexem, 0] = str(4)
                Lexems[CurrentLexem, 1] = str(main.CurNum + 1)

                main.Numbers[main.CurNum] = BufNum[0:len(BufNum) - 1]
                CurrentLexem += 1
                main.CurNum += 1

            elif main.WordChar[main.CurPosition] == 'e' or
main.WordChar[main.CurPosition] == 'E' or main.WordChar[
                main.CurPosition] == '.':
                main.CurPosition -= 1
                BufNum = BufNum[0:len(BufNum) - 1]
                CurrentCondition = "EXP"

            else:
                ErrorLexer(108)

            main.CurPosition += 1


        elif CurrentCondition == "DELIMETER": #Изменил для работы
```

```
            if main.WordChar[main.CurPosition] == ":" and
main.WordChar[main.CurPosition + 1] == "=":
                Lexems[CurrentLexem, 0] = str(
                    GetLexems("".join(main.WordChar[main.CurPosition:
main.CurPosition + 2]))[0])

                Lexems[CurrentLexem, 1] = str(
                    GetLexems("".join(main.WordChar[main.CurPosition:
main.CurPosition + 2]))[1])

                main.CurPosition += 2

                CurrentLexem += 1

                CurrentCondition = "H"

                continue

            if main.WordChar[main.CurPosition] == '(' or
main.WordChar[main.CurPosition] == ')' or main.WordChar[

                main.CurPosition] == ':' or main.WordChar[main.CurPosition] ==
'[' or main.WordChar[

                main.CurPosition] == ']' or main.WordChar[main.CurPosition] ==
',' or main.WordChar[

                main.CurPosition] == ';' or main.WordChar[main.CurPosition] ==
'{' or main.WordChar[

                main.CurPosition] == " ":

                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                # Lexems[CurrentLexem][1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])
```

```
                Lexems[CurrentLexem, 0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                Lexems[CurrentLexem, 1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])

                main.CurPosition += 1

                CurrentLexem += 1

            elif main.WordChar[main.CurPosition] == ':' or
main.WordChar[main.CurPosition] == '=':

                if main.Sost == True:

                    if main.WordChar[main.CurPosition - 1] == ':' and
main.WordChar[main.CurPosition] == '=':

                        CurrentCondition = "H"

                        Lexems[CurrentLexem, 0] = str(GetLexems(

                            str(str(main.WordChar[main.CurPosition - 1]) +
str(main.WordChar[main.CurPosition])))[0])

                        Lexems[CurrentLexem, 1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])

                        main.CurPosition += 1

                        CurrentLexem += 1

                    elif main.WordChar[main.CurPosition - 1] == '=' and
main.WordChar[main.CurPosition] == ':':

                        ErrorLexer(228)

                else:

                    main.Sost = True
```

```
                    main.CurPosition += 1

                    print(main.WordChar[main.CurPosition - 1])


            elif main.WordChar[main.CurPosition] == '/*':

                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                # Lexems[CurrentLexem][1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])

                Lexems[CurrentLexem, 0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                Lexems[CurrentLexem, 1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])

                main.CurPosition += 1

                CurrentLexem += 1


            elif main.WordChar[main.CurPosition] == '*/':

                CurrentCondition = "H"

                # Lexems[CurrentLexem][0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                # Lexems[CurrentLexem][1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])

                Lexems[CurrentLexem, 0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                Lexems[CurrentLexem, 1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])
```

```
                main.CurPosition += 1

                CurrentLexem += 1


            elif main.WordChar[main.CurPosition].isdigit() or
main.WordChar[main.CurPosition].isalpha():

                if main.Sost == True:
                    Lexems[CurrentLexem, 0] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[0])

                    Lexems[CurrentLexem, 1] =
str(GetLexems(str(main.WordChar[main.CurPosition]))[1])

                    # Lexems[CurrentLexem][0] =
str(GetLexems(main.WordChar[main.CurPosition - 1])[0])

                    # Lexems[CurrentLexem][1] =
str(GetLexems(main.WordChar[main.CurPosition - 1])[1])

                    CurrentLexem += 1

                CurrentCondition = "H"

            else:

                ErrorLexer(110)

        # print("ПОЧТИ ВСЁ ХОРОШО !")

    # print("ПОЧТИ ВСЁ ХОРОШО !")

    if main.IsLetter == False:
        ErrorLexer(111)

    CountCurLexem = 0
    while CountCurLexem < CurrentLexem:
        tmp = str(Lexems[CountCurLexem, 0]) + " " + str(Lexems[CountCurLexem,
1]) + "\n"
```

```
        GenerateFiles.FileLexems.write(tmp)
        CountCurLexem += 1
    GenerateFiles.FileLexems.close()

    CountNumbers = 0
    while CountNumbers < len(main.Numbers):
        if main.Numbers[CountNumbers] != None:
            tmp = str(main.Numbers[CountNumbers]) + " " + str(CountNumbers +
1) + "\n"
            GenerateFiles.FileNumbers.write(tmp)
        CountNumbers += 1
    GenerateFiles.FileNumbers.close()

    # print("main.Identificators: ", main.Identificators)

    CountIdentificators = 0
    while CountIdentificators < len(main.Identificators):
        if main.Identificators[CountIdentificators] != None:
            tmp = str(main.Identificators[CountIdentificators] + " " +
str(CountIdentificators + 1) + "\n")
            GenerateFiles.FileIdentificators.write(tmp)
        CountIdentificators += 1
    GenerateFiles.FileIdentificators.close()


def ErrorLexer(NumOfError):
    if NumOfError == 101:
        print("Error: ", NumOfError, "Ожидалось число.")
        exit(0)
    elif NumOfError == 102:
        print("Error: ", NumOfError, "Ожидалось число '}'")
        exit(0)
    elif NumOfError == 103:
        print("Error: ", NumOfError, "Ожидалось число или e или E или. или b
или B.")
        exit(0)
    elif NumOfError == 104:
        print("Error: ", NumOfError, "Ожидалось число или e или E или. или o
или O.")
        exit(0)
    elif NumOfError == 105:
```

```
        print("Error: ", NumOfError, "Ожидалось отсутствие точки после e.")
        exit(0)
    elif NumOfError == 106:
        print("Error: ", NumOfError, "Ожидалось e или E")
        exit(0)
    elif NumOfError == 107:
        print("Error: ", NumOfError, "Ожидалось i после d.")
        exit(0)
    elif NumOfError == 108:
        print("Error: ", NumOfError, "Ожидалось число или e или E или. или d
или D или h или H.")
        exit(0)
    elif NumOfError == 109:
        print("Error: ", NumOfError, "")
        exit(0)
    elif NumOfError == 110:
        print("Error: ", NumOfError, "Ожидался разделитель.")
        exit(0)
    elif NumOfError == 111:
        print("Error: ", NumOfError, "Ожидалась буква для описания cc.")
        exit(0)
    elif NumOfError == 112:
        print("Error: ", NumOfError, "Ожидалось продолжение действительного
числа.")
        exit(0)
    elif NumOfError == 113:
        print("Error: ", NumOfError, "Ожидалось продолжение действительного
числа.")
        exit(0)
    elif NumOfError == 114:
        print("Error: ", NumOfError,
              "Ожидалось D или d или B или b или H или h или O или o или . или
E или e или число.")
        exit(0)
    elif NumOfError == 228:
        print("Error: ", NumOfError, ":= != =:")
        exit(0)


def GetNum(BufNum, NumberOfSystem):
```

```
    if BufNum[len(BufNum) - 1] == 'o' or BufNum[len(BufNum) - 1] == 'O' or
BufNum[len(BufNum) - 1] == 'h' or BufNum[
        len(BufNum) - 1] == 'H' or BufNum[len(BufNum) - 1] == 'b' or
BufNum[len(BufNum) - 1] == 'B':
        BufNum = BufNum[0:len(BufNum) - 1]
    if NumberOfSystem == "HEX":
        BufNum = int(BufNum)
        BufNum = hex(BufNum)
        return str(BufNum)
    elif NumberOfSystem == "BIN":
        BufNum = int(BufNum)
        BufNum = bin(BufNum)
        return str(BufNum)
    elif NumberOfSystem == "OCT":
        BufNum = int(BufNum)
        BufNum = oct(BufNum)
        return str(BufNum)
    return "0"


# Получаем Лексемы
def GetLexems(Word):
    cur = 1

    # print("Word", Word)
    # print(main.KeyWord)
    # print("Delimeters: ", main.Delimeters)

    for s in main.KeyWord:

        # print("word == s: ", Word, "==", s)
        # print(bool(Word == s))

        if Word == s:
            # print("Debug: ", 1, cur)
            return (1, cur)
        cur += 1
    cur = 1
    for s in main.Delimeters:

        # print("Word == s: ", Word, "==", s)
```

```
        # print(bool(Word == s))

        if Word == s:
            # print("Debug: ", 2, cur)
            return (2, cur)
        cur += 1
    return (0, 0)
```

Приложение Б

Код синтаксического анализатора (с семантическими проверками)

```
import main
import GenerateFiles
import LexModule
from isHex import isHex
import numpy as np

Num = []

Val = []

NewFileNumbers = open("Numbers.txt", 'r')

NewFileIdentificators = open("Identificators.txt", 'r')

NewFileLexems = open("FinalLexems.txt", 'r')

DeclaredIdentificators = []


def ErrorParser(ErrorOfParser):
    if ErrorOfParser == 201:
        print("Error: ", ErrorOfParser, "ожидался оператор группы умножения")
        exit(0)
    elif ErrorOfParser == 201: # 202
        print("Error:", ErrorOfParser, "Ожидалось ';' после ключевого слова
'begin' или перенос строки до ключевого слова 'begin' или объявлен ещё один
тип данных или другая ошибка")
        exit(0)
    elif ErrorOfParser == 203:
        print("Error: ", ErrorOfParser, "Ожидалось ';' или перенос строки")
        exit(0)
    elif ErrorOfParser == 201: #204
        print("Error:", ErrorOfParser, "ожидалось 'end.'")
        exit(0)
    elif ErrorOfParser == 205:
        print("Error: ", ErrorOfParser, "ожидался идентификатор")
        exit(0)
    elif ErrorOfParser == 206:
```

```
    print("Error: ", ErrorOfParser)
    exit(0)
elif ErrorOfParser == 207:
    print("Error: ", ErrorOfParser, "ожидался тип перемнной")
    exit(0)
elif ErrorOfParser == 208:
    print("Error: ", ErrorOfParser, "ожидалась ']'")
    exit(0)
elif ErrorOfParser == 209:
    print("Error: ", ErrorOfParser, "ожидалось ключевое слово 'as'")
    exit(0)
elif ErrorOfParser == 210:
    print("Error: ", ErrorOfParser, "ожидалось ключевое слово 'then'")
    exit(0)
elif ErrorOfParser == 211:
    print("Error: ", ErrorOfParser, "ожидалось ключевое слово 'to'")
    exit(0)
elif ErrorOfParser == 212:
    print("Error: ", ErrorOfParser, "ожидалось выражение")
    exit(0)
elif ErrorOfParser == 213:
    print("Error: ", ErrorOfParser, "Ожидался do")
    exit(0)
elif ErrorOfParser == 214:
    print("Error: ", ErrorOfParser, "ожидался оператор")
    exit(0)
elif ErrorOfParser == 215:
    print("Error: ", ErrorOfParser, "ожидалась '('")
    exit(0)
elif ErrorOfParser == 216:
    print("Error: ", ErrorOfParser, "ожидалась операция группы сложения")
    exit(0)
elif ErrorOfParser == 217:
    print("Error: ", ErrorOfParser, "ожидалась ')'")
    exit(0)
elif ErrorOfParser == 218:
    print("Error: ", ErrorOfParser, "Неопознанная ошибка")
    exit(0)
elif ErrorOfParser == 219:
    print("Error: ", ErrorOfParser, "Ожидалась операция группы отношения")
    exit(0)
```

```
    elif ErrorOfParser == 220:
        print("Error: ", ErrorOfParser, "Неопознанная ошибка")
        exit(0)
    elif ErrorOfParser == 221:
        print("Error: ", ErrorOfParser, "Ожидалась унарная операция")
        exit(0)
    elif ErrorOfParser == 301:
        print("Error: ", ErrorOfParser, "повторное объявление переменной")
        exit(0)
    elif ErrorOfParser == 302:
        print("Error: ", ErrorOfParser, "использование необъявленной
переменной")
        exit(0)


def Parser():
    Filling()
    Prog()


def add():

    #print("Add: DeclaredIdentificators:", DeclaredIdentificators)

    DeclaredIdentificators.append(Val[main.CurLexLexem - 1])

    #print("Add: DeclaredIdentificators:", DeclaredIdentificators)


def Check():
    #print(DeclaredIdentificators)
    for i in DeclaredIdentificators:
        print("i", i)
        #  print("Type(Num[main.CurLexLexem-1])", type(Num[main.CurLexLexem-
1]))

        # print("main.Identificators[int(Val[main.CurLexLexem-1])-1])",
type(main.Identificators[int(Val[main.CurLexLexem-1])-1]))

        #  print(main.Identificators)
        #  print(type(main.Identificators))
```

```
        if main.Identificators[int(Val[main.CurLexLexem-1])-1] ==
str(main.Identificators[int(i)-1]) and Num[main.CurLexLexem-1] == "3":
            return True
    return False


"""
<ОПИСАНИЕ>::= <ТИП> <ИДЕНТИФИКАТОР> {, <ИДЕНТИФИКАТОР> }
!!!! ПРОВЕРИТЬ
"""


def Discription():
    if Equals("/*"):
        Comment()

    # print("main.CurLexLexemWord ", main.CurLexLexemWord)

    if Tipisation():

        #Get_Lexem()

        if Check():
            ErrorParser(301)
        else:
            add()

        Get_Lexem()

        if Equals("/*"):
            Comment()

        #  print("144 main.CurLexLexemWord", main.CurLexLexemWord)
        #  print("145 ", Num[main.CurLexLexem - 1])

        while Equals(","):

            if Equals("/*"):
                Comment()
```

```
            Get_Lexem()

            if not(ID()):
                ErrorParser(205)

            else:
                if Check():
                    ErrorParser(301)
                else:
                    add()

            Get_Lexem()

    else:
        ErrorParser(206)


def ID():

    # print("ID: Num[main.CurLexLexem - 1]", Num[main.CurLexLexem - 1])

    if Num[main.CurLexLexem - 1] == "3":
        return True
    else:
        return False


def Numer():
    if Num[main.CurLexLexem - 1] == "4":
        return True
    else:
        return False


# StrArgument -> S; READY
def Equals(StrArgument):
    counter = 1
    #  print("номер:", main.CurLexLexem - 1)
    # print("Equals: Num[main.CurLexLexem - 1]", Num[main.CurLexLexem - 1])

    if Num[main.CurLexLexem - 1] == "1":
```

```
        for i in main.KeyWord:
            if i == StrArgument:
                if str(counter) == Val[main.CurLexLexem - 1]:
                    # print("Equals: Val[main.CurLexLexem - 1]",
Val[main.CurLexLexem - 1])
                    return True
                else:
                    return False
            counter += 1
    elif Num[main.CurLexLexem - 1] == "2":
        for i in main.Delimeters:
            if i == StrArgument:
                if str(counter) == Val[main.CurLexLexem - 1]:
                    # print("Equals: Val[main.CurLexLexem - 1]",
Val[main.CurLexLexem - 1])
                    return True
                else:
                    return False
            counter += 1
    elif Num[main.CurLexLexem - 1] == "3":
        for i in main.Identificators:
            if i == StrArgument:
                if str(counter) == Val[main.CurLexLexem - 1]:
                    # print("Equals: Val[main.CurLexLexem - 1]",
Val[main.CurLexLexem - 1])
                    return True
                else:
                    return False
            counter += 1

    elif Num[main.CurLexLexem - 1] == "4":
        for i in main.Numbers:
            if i == StrArgument:
                if str(counter) == Val[main.CurLexLexem - 1]:
                    # print("Equals: Val[main.CurLexLexem - 1]",
Val[main.CurLexLexem - 1])
                    return True
                else:
                    return False
            counter += 1
    else:
```

```
        return False

    return False

def Filling():
    CurrentLexem = NewFileLexems.readlines()

    count = 0
    while count < len(CurrentLexem):
        CurrentLexem[count] = CurrentLexem[count][:-1]
        count += 1

    count = 0
    LenCurrentLexems = len(CurrentLexem)
    while count < LenCurrentLexems:
        tmp = CurrentLexem[count][:1]
        Num.append(tmp)

        tmp = CurrentLexem[count][2:]
        Val.append(tmp)

        count += 1

    #  print(Num)
    #  print(Val)

"""
COMPAREOPERAND
<СОСТАВНОЙ>::= «[» <ОПЕРАТОР> { ( : | ПЕРЕВОД СТРОКИ) <ОПЕРАТОР> } «]»
READY
"""


def CompareOperand():
    while Equals(":") or Equals("\n"):
        if Equals("/*"):
            Comment()

        Get_Lexem()

        Oper()
```

```
    if Equals("/*"):
        Comment()

    if not(Equals("}")):
        ErrorParser(208)

    Get_Lexem()


"""
ASSIGNOPER
<ПРИСВАИВАНИЕ>::= <ИДЕНТИФИКАТОР> as <ВЫРАЖЕНИЕ>
READY
"""


def AssignOper():
    Get_Lexem()
    if Equals("/*"):
        Comment()

    if (Check()):
        ErrorParser(302)

    Get_Lexem()

    if Equals("/*"):
        Comment()

    if (Equals("as")):
        ErrorParser(209)

    #Get_Lexem()

    Expression()


"""
IFOPERAND
<ФИКСИРОВАННЫЙ_ЦИКЛ>::= for <ПРИСВАИВАНИЕ> to <ВЫРАЖЕНИЕ> do <ОПЕРАТОР>
```

```
READY
"""


def IfOperand():
    if Equals("/*"):
        Comment()

    Get_Lexem()

    Expression()

    if Equals("/*"):
        Comment()

    if not(Equals("then")):
        ErrorParser(210)

    Get_Lexem()

    Oper()

    if Equals("{"):
        Comment()

    if Equals("else"):
        Get_Lexem()
        Oper()


"""
FORCYCLE
<ФИКСИРОВАННЫЙ_ЦИКЛ>::= for <ПРИСВАИВАНИЕ> to <ВЫРАЖЕНИЕ> do <ОПЕРАТОР>
READY
"""


def ForCycle():
    if Equals("/*"):
        Comment()
```

```
    Get_Lexem()

    if ID():
        AssignOper()

    if Equals("/*"):
        Comment()

    if not(Equals("to")):
        ErrorParser(211)

    Get_Lexem()

    if Equals("/*"):
        Comment()

    if ID() or Numer() or Equals("true") or Equals("false") or Equals("not")
or Equals("(") or Equals("as"):
        if ID() and not(Check()):
            ErrorParser(302)
        Expression()
    else:
        ErrorParser(212)

    if Equals("/*"):
        Comment()

    if not(Equals("do")):
        ErrorParser(213)

    Get_Lexem()

    Oper()


"""
WHILECYCLE
<УСЛОВНЫЙ_ЦИКЛ>::= while <ВЫРАЖЕНИЕ> do <ОПЕРАТОР>
READY
"""
```

```
def WhileCycle():
    if Equals("/*"):
        Comment()

    if ID() or Numer() or Equals("true") or Equals("true") or Equals("not") or
Equals("(") or Equals("as"):
        if ID() and not(Check()):
            ErrorParser(302)
        Expression()
    else:
        ErrorParser(212)

    Get_Lexem()

    if Equals("/*"):
        Comment()

    if not(Equals("do")):
        ErrorParser(213)

    if Equals("[") or Equals("if") or Equals("for") or Equals("while") or
Equals("read") or Equals("write") or ID():
        Oper()
    else:
        ErrorParser(214)


"""
INPUT
<ВВОД>::= read «(»<ИДЕНТИФИКАТОР> {, <ИДЕНТИФИКАТОР> } «)»
READY
"""


def Input():

    if Equals("/*"):
        Comment()

    Get_Lexem()
```

```
    if not(Equals("(")):
        ErrorParser(215)

    Get_Lexem()

    if Equals("/*"):
        Comment()

    if not(ID()):
        ErrorParser(205)

    Get_Lexem()

    if Equals("/*"):
        Comment()

    while Equals(","):

        if Equals("/*"):
            Comment()

        Get_Lexem()

        if ID() and not(Check()):
            ErrorParser(302)

        Get_Lexem()

    if not Equals(")"):
        ErrorParser(217)
    Get_Lexem()


"""
OUTPUT
<ВЫВОД>::= write «(»<ВЫРАЖЕНИЕ> {, <ВЫРАЖЕНИЕ> } «)»
READY
"""
```

```
def Output():
    if Equals("/*"):
        Comment()
    Get_Lexem()

    if Equals("/*"):
        Comment()

    if not(Equals("(")):
        ErrorParser(215)
    #Get_Lexem()

    if ID() or Numer() or Equals("true") or Equals("false") or Equals("not")
or Equals("(" or Equals("as")):
        if ID() and not(Check()):
            ErrorParser(302)
        Expression()
        if Equals("/*"):
            Comment()
        while Equals(","):

            if Equals("/*"):
                Comment()

            Get_Lexem()

            if Equals("/*"):
                Comment()

            if ID() or Numer() or Equals("true") or Equals("false") or
Equals("not") or Equals("(") or Equals("as"):
                if ID() and not(Check()):
                    ErrorParser(302)
                Expression()
            else:
                ErrorParser(212)

    else:
        ErrorParser(212)

    if Equals("/*"):
```

```
        Comment()

    if not(Equals(")")):
        ErrorParser(217)

    Get_Lexem()


"""
NUMBER
<ЧИСЛО>:: =  <ЦЕЛОЕ> | <ДЕЙСТВИТЕЛЬНОЕ>
READY
"""


def Number():
    if Equals("/*"):
        Comment()
    if not(Numer()):
        ErrorParser(219)
    Get_Lexem()


"""
TYPE
<ТИП>::= INT | FLOAT | BOOL
READY
"""


def Tipisation():
    if Equals("/*"):
        Comment()
    if not(Equals("integer")) and not(Equals("real")) and
not(Equals("boolean")):
        ErrorParser(207)
    Get_Lexem()
    return  True


"""
```

```
RATIO
<ОПЕРАЦИИ_ГРУППЫ_ОТНОШЕНИЯ>:: = NE | EQ | LT | LE | GT | GE
READY
"""


def Ratio():
    if Equals("/*"):
        Comment()
    if not(Equals("<>") or Equals("=") or Equals("<") or Equals("<=>") or
Equals(">") or Equals(">=")):
        ErrorParser(219)
    Get_Lexem()


"""
ADDITION
<ОПЕРАЦИИ_ГРУППЫ_СЛОЖЕНИЯ>:: = + | - | OR
READY
"""


def Addition():
    if Equals("/*"):
        Comment()
    #  print("main.CurLexLexemWord", main.CurLexLexemWord)

    if not(Equals("+") or Equals("-") or Equals("or")):
        ErrorParser(216)
    Get_Lexem()


"""
MULTIPLICATION
<ОПЕРАЦИИ_ГРУППЫ_УМНОЖЕНИЯ>::= * | / | AND
READY
"""


def Multiplicate():
    if Equals("/*"):
```

```
        Comment()
    if not(Equals("*") or Equals("/") or Equals("and")):
        ErrorParser(201)
    Get_Lexem()


"""
UNAR
<УНАРНАЯ ОПЕРАЦИЯ>::= ~
READY
"""


def Unar():
    if Equals("/*"):
        Comment()
    if not (Equals("not")):
        ErrorParser(223)
    Get_Lexem()


"""
COMMENT
<КОММЕНТАРИЙ>::="{"<СИМВОЛ>"}"
READY
"""


def Comment():
    while not (Equals("*/")):
        Get_Lexem()
    Get_Lexem()


"""
OPER
<ОПЕРАТОР>::=     <СОСТАВНОЙ> | <ПРИСВАИВАНИЕ> | <УСЛОВНЫЙ>
|<ФИКСИРОВАННЫЙ_ЦИКЛ> | <УСЛОВНЫЙ_ЦИКЛ> | <ВВОД> |<ВЫВОД>
"""
```

```
def Oper():
    if Equals("/*"):
        Comment()

    if Equals("["):
        Comment()
    elif Equals("as"):
        AssignOper()
    elif Equals("if"):
        IfOperand()

    elif Equals("for"):
        ForCycle()

    elif Equals("while"):
        WhileCycle()

    elif Equals("read"):
        Input()

    elif Equals("write"):
        Output()

    elif ID():
        AssignOper()



def Get_Lexem():
    counter = 1

    #  print("Get_Lexem: main.CurLexLexem", main.CurLexLexem)
    #  print("Get_Lexem: Num[main.CurLexLexem]", Num[main.CurLexLexem] )
    #  print("Get_Lexem: Val[main.CurLexLexem]", Val[main.CurLexLexem])

    if Num[main.CurLexLexem] == "1":
        for s in main.KeyWord:
            if str(counter) == Val[main.CurLexLexem]:
                main.CurLexLexemWord = s
                # print(" main.CurLexLexemWord", main.CurLexLexemWord)
            counter += 1
    elif Num[main.CurLexLexem] == "2":
```

```
        for s in main.Delimeters:
            if str(counter) == Val[main.CurLexLexem]:
                main.CurLexLexemWord = s
                # print(" main.CurLexLexemWord", main.CurLexLexemWord)
            counter += 1
    elif Num[main.CurLexLexem] == "3":
        for s in main.Identificators:
            if str(counter) == Val[main.CurLexLexem]:
                main.CurLexLexemWord = s
                # print(" main.CurLexLexemWord", main.CurLexLexemWord)
            counter += 1
    elif Num[main.CurLexLexem] == "4":
        for s in main.Numbers:
            if str(counter) == Val[main.CurLexLexem]:
                main.CurLexLexemWord = s
            counter += 1
    else:
        ErrorParser(201)
    #except:
        #ErrorParser(204)
    main.CurLexemNumber = main.CurLexLexem
    main.CurLexLexem += 1


"""
СТРУКТУРА ПРОГРАММЫ:
//<ПРОГРАММА> = PROGRAM VAR <ОПИСАНИЕ> BEGIN <ОПЕРАТОР> {; <ОПЕРАТОР>} end.
«{» {/ (<описание> | <оператор>) ; /} «}»
"""


def Prog():
    Get_Lexem()

    if (Equals("{")):
            Get_Lexem()

            if Equals("\n"):
                Get_Lexem()

            if Equals("/*"):
```

```
                Comment()

            if Equals("integer") or Equals("boolean") or Equals("real"):
                Discription()

            if Equals("\n"):
                Get_Lexem()

            if Equals("[") or Equals("if") or Equals("for") or Equals("while")
or Equals("read") or Equals("write") or ID():
                Oper()

            if Equals("}"):
                print("конец программы")
            else:
                ErrorParser(202)

    if Equals("/*"):
        Comment()

    if not (Equals("}")):
        ErrorParser(204)

'''
"""
СТРУКТУРА ПРОГРАММЫ:
//<ПРОГРАММА> = PROGRAM VAR <ОПИСАНИЕ> BEGIN <ОПЕРАТОР> {; <ОПЕРАТОР>} end.
"""


def Prog():
    Get_Lexem()

    if Equals("program"):
        Get_Lexem()

        if Equals("var"):
            Get_Lexem()

            if Equals("\n"):
                Get_Lexem()
```

```
            if Equals("{"):
                Comment()

            if Equals("int") or Equals("bool") or Equals("float"):
                Discription()

            if Equals("\n"):
                Get_Lexem()

            if Equals("begin"):
                Get_Lexem()

            if Equals("\n"):
                Get_Lexem()

            if Equals("[") or Equals("if") or Equals("for") or Equals("while")
or Equals("read") or Equals("write") or ID():
                Oper()

            #Get_Lexem()

            while Equals(";"):
                Get_Lexem()

                if Equals("\n"):
                    Get_Lexem()

                if Equals("{"):
                    Comment()

                if Equals("\n"):
                    Get_Lexem()

                if Equals("[") or Equals("if") or Equals("for") or
Equals("while") or Equals("read") or Equals("write") or ID():
                    Oper()

            if Equals("end."):
                print("конец программы")
            else:
```

```
                ErrorParser(202)

    if Equals("{"):
        Comment()

    if not (Equals("end.")):
        ErrorParser(204)


'''

"""
EXPRESSION
<ВЫРАЖЕНИЕ>:: =   <ОПЕРАНД>{ <ОПЕРАЦИИ_ГРУППЫ_ОТНОШЕНИЯ> <ОПЕРАНД> }
"""


def Expression():
    Get_Lexem()

    if Equals("/*"):
        Comment()

    if Numer() or ID() or Equals("true") or Equals("false") or Equals("not")
or Equals("(") or Equals("as"):

        if Equals("/*"):
            Comment()

        if ID() and (Check()):
            ErrorParser()

        Operand()

    if Equals("/*"):
        Comment()

    if Equals("<>") or Equals("<") or Equals("<=") or Equals(">") or
Equals(">=") or Equals("="):
        Ratio()
```

```
        if Equals("/*"):
            Comment()

        if ID() or Numer() or Equals("true") or Equals("false") or
Equals("not") or Equals("(") or Equals("as"):

            if Equals("/*"):
                Comment()

            if ID() and not(Check()):
                ErrorParser(302)

            Operand()

        else:
            ErrorParser(214)

    #Get_Lexem()


"""
OPERAND
<ОПЕРАНД>::= <СЛАГАЕМОЕ> {<ОПЕРАЦИИ_ГРУППЫ_СЛОЖЕНИЯ> <СЛАГАЕМОЕ>}
"""


def Operand():
    #  print("Operand main.CurLexLexemWord", main.CurLexLexemWord)

    if Equals("/*"):
        Comment()

    if ID() or Numer() or Equals("true") or Equals("false") or Equals("as")
or Equals("not") or Equals("/*"):
        if ID() and not(Check()):
            ErrorParser(302)
        Summand()

    if Equals("/*"):
        Comment()
```

```
    if Equals("+") or Equals("-") or Equals("or"):
        Addition()

        if Equals("/*"):
            Comment()

        if ID() or Numer() or Equals("true") or Equals("false") or
Equals("not") or Equals("(") or Equals("as"):
            if ID() and not(Check()):
                ErrorParser(302)
            Summand()
        else:
            ErrorParser(214)


"""
SUMMAND
<СЛАГАЕМОЕ>::=    <МНОЖИТЕЛЬ> {<ОПЕРАЦИИ_ГРУППЫ_УМНОЖЕНИЯ> <МНОЖИТЕЛЬ>}
"""


def Summand():
    if Equals("/*"):
        Comment()

    if ID() or Numer() or Equals("true") or Equals("false") or Equals("not")
or Equals("(") or Equals("as"):
        if ID() and not(Check()):
            ErrorParser(302)
        Multiply()

    else:
        ErrorParser(214)

    if Equals("/*"):
        Comment()

    if Equals("*") or Equals("/") or Equals("and"):

        Multiplicate()
```

```
        if Equals("/*"):
            Comment()

        if ID() or Numer() or Equals("true") or Equals("false") or
Equals("not") or Equals("(") or Equals("as"):
            if ID() and not(Check()):
                ErrorParser(302)
            Multiply()
        else:
            ErrorParser(214)


"""
MULTIPLY
<МНОЖИТЕЛЬ>::=    <ИДЕНТИФИКАТОР> | <ЧИСЛО> | <ЛОГИЧЕСКАЯ_КОНСТАНТА>
|<УНАРНАЯ_ОПЕРАЦИЯ> <МНОЖИТЕЛЬ> | «(»<ВЫРАЖЕНИЕ>«)»
"""


def Multiply():
    if Equals("/*"):
        Comment()

    if Equals("("):

        Get_Lexem()

        Expression()

        if Equals("/*"):
            Comment()

        if not(Equals(")")):
            ErrorParser(217)

    if Equals("/*"):
        Comment()

    elif Equals("not"):
        Unar()
```

```
Get_Lexem()
```