Министерство образования и науки РФ

Санкт-Петербургский Политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Направление 02.03.01 «Математика и компьютерные науки»

Дисциплина «Математическая логика и теория автоматов»

Отчёт по лабораторной работе №1

«Построение лексического анализатора»
Вариант 3

Студент: Башарина Е.А., гр. 3530201/90101

Преподаватель: Востров А.В.

Санкт-Петербург — 2022

Содержание

Введение

Лексический анализ текста - это процесс, в ходе которого некоторый текст структурируется и проверяется на соответствие некоторому *языку*. Он включает в себя распознавание значащих последовательностей (лексем), таких как ключевые слова, идентификаторы, константы и символы арифметических операций. В результате лексического анализа входной текст проверяется на соответствие правилам определённого языка, а все обнаруженные лексемы группируются по типу и структурируются в виде таблицы.
Данный отчёт содержит описание выполнения лабораторной работы, в ходе которой был разработан лексический анализатор для языка, определённого вариантом задания. Лабораторная работа была написана на языке Python в среде программирования PyCharm.

Цель работы: написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений.

1 Математическое описание

1.1 Лексический анализ

Лексема — это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п. Лексемы могут состоять из нескольких символов - например, служебные и ключевые слова, идентификаторы.

Лексический анализатор — это программа, которая читает поток символов, составляющих исходный текст, и группирует эти символ в значащие последовательности (лексемы). Лексический анализатор распознаёт лексемы в программе, а также выбрасывает комментарии и пробельные символы.

1.2 Особенности входного языка

Ограничения

- Текст на входном языке задается в виде символьного (текстового) файла.
- Программа должна выдавать сообщения о наличии во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.
- Длина идентификатора и строковых констант ограничена 16 символами.
- Программа должна допускать наличие комментариев неограниченной длины во входном файле (форма комментариев выбирается самостоятельно).

Лексемы

Входной язык содержит:

- операторы условия **if ... then ... else** и **if ... then**, разделенные символом **;** (точка с запятой),
- идентификаторы,
- знаки сравнения $<$, $>$, $=$,
- десятичные числа с плавающей точкой (в обычной и логарифмической форме)
- знак присваивания (**:=**).

2 Реализация

Для реализации задания были написаны два класса: **Analyser**, реализующий анализ входного текста, и **IO_worker**, организующий работу с файлами. Пользовательский ввод-вывод реализован в файле **main.py**. Подробное описание этих файлов представлено ниже.

2.1 Analyser.py

В конструкторе класса Analyser задаются допустимые символы входного языка и ограничения, которые он накладывает (для проверки идентификаторов и чисел использованы регулярные выражения). Кроме того, из входного текста удаляются все комментарии.
Вход: text - строка символов, считанная из файла
Выход: конструктор не возвращает никакого значения

```
class Analyser:
    def __init__(self, text):
        self.text = [] # input text to work with
        self.idents = {} # dictionary of identificators
        self.id_idents = 1 # counter of identificators
        self.kws = {} # dictionary of keywords
        self.id_kws = 1 # counter of keywords
        self.remove_comments(text.split('\n'))
        self.table = [] # table of tokens
        self.keywords = ['if', 'then', 'else'] # list of possible keywords
        self.compare = ['>', '<', '='] # list of possible compare signs
        self.equal = ':=' # equality sign
        self.sep = ';' # separator
        self.identif = r'[A-Za-z]{1,16}' # regExp for identificators
        self.number = r'[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][-+]?\d+)?' #regExp for numbers
```

Метод analyse() производит анализ входного текста: до тех пор, пока текст не пуст, он разделяется на части head (последовательность присвоений значений идентификаторам, не относящаяся к условным конструкциям) и tail (часть с условными конструкциями). Для head вызывается метод basic_cycle(), а для tail - метод if_then_cycle(), которые удаляют из входного текста все проверенные цепочки. В результате в поле self.text остаётся только часть текста, следующая за первой встреченной условной конструкцией, и для нее метод analyse() вызывается рекурсивно.
Вход: метод не принимает на вход никакого значения
Выход: метод не возвращает никакого значения

```
    def analyse(self):
        if len(self.text) == 0:
            return
        else:
            head = []
            tail = self.text.copy()
            for word in self.text:
                if word not in self.keywords:
                    head.append(word)
                    tail.pop(0)
                else:
```

```
                break
        if len(head) != 0:
            self.basic_cycle(head)
        if len(tail) != 0:
            self.text = self.if_then_cycle(tail)
        else:
            self.text = []
        self.analyse()
```

Метод basic_cycle() реализует проверку последовательности присвоений вида «identificator := identificator» либо «identificator := number». В случае несовпадения какой-либо лексемы (возникновения AssertionError) вызывается метод handle() для идентификаторов и чисел, либо же выводится на экран ошибка о некорректном символе. При этом в таблицу лексем добавляется каждая встреченная корректная лексема, а также её тип и значение.
Вход: text - список символов для анализа
Выход: метод не возвращает никакого значения

```
    def basic_cycle(self, text):
        first, second, third = 0, 1, 2
        while first != len(text):
            try:
                assert re.fullmatch(self.identif, text[first])
                if text[first] not in self.idents.keys():
                    self.idents[text[first]] = self.id_idents
                self.id_idents += 1
                self.table.append(text[first]+'Идентификатор;;'+\
        str(self.idents[text[first]]))
            except AssertionError:
                self.handle(text[first])
            try:
                assert text[second] == self.equal
                self.table.append(text[second] + 'Знак; присваивания')
            except AssertionError:
                print('Invalid symbol: '+text[second]+', `:=` expected')
            try:
                assert re.fullmatch(self.identif, text[third]) or\
          re.fullmatch(self.number, text[third])
                if re.fullmatch(self.identif, text[third]):
                    if text[third] not in self.idents.keys():
                        self.idents[text[third]] = self.id_idents
                    self.id_idents += 1
                    self.table.append(text[third] + 'Идентификатор;;' +\
        str(self.idents[text[third]]))
                else:
                    self.table.append(text[third] + 'Вещественная; константа;'\
        + text[third])
            except AssertionError:
                self.handle(text[third])
            first += 3
            second += 3
            third += 3
```

Метод if_then_cycle() предназначен для проверки условной конструкции. Для этого поступивший на вход текст проверяется на соответствие следующему шаблону:

1. if
2. identificator
3. compare sign
4. identificator либо number
5. последовательность присвоений вида «identificator := identificator» либо «identificator := number», оканчивающаяся разделителем (;)
6. then
7. последовательность присвоений вида «identificator := identificator» либо «identificator := number», оканчивающаяся разделителем (;)

   ОПЦИОНАЛЬНО:

8. else
9. последовательность присвоений вида «identificator := identificator» либо «identificator := number», оканчивающаяся разделителем (;)

Для пунктов 5, 7 и 9 используетя вложенный метод extract_base(), извлекающий последовательность присвоений, которая позже передаётся для валидации в метод basic_cycle(). После того, как шаблон считывается целиком, метод возвращает остаток исходного текста для дальнейшего анализа. Аналогично методу basic_cycle(), в ходе валидации корректные лексемы добавляются в таблицу, а некорректные выводятся на экран с комментарием.
Вход: text - список символов для анализа
Выход: модифицированный список символов (удалены элементы, уже прошедшие процедуру проверки)

```
    def if_then_cycle(self, text):
        def extract_base():
            base = []
            w = text.pop(0)
            while w[-1] != ';':
                base.append(w)
                w = text.pop(0)
            base.append(w[:-1])
            self.basic_cycle(base)

        word = text.pop(0)
        try:
            assert word == 'if'
            if word not in self.kws.keys():
                self.kws[word] = self.id_kws
            self.id_kws += 1
            self.table.append(word + 'Ключевое; словоX;' + str(self.kws[word]))
        except AssertionError:
            print('Invalid keyword: '+word+', `if` expected')
        word = text.pop(0)
        try:
```

```
            assert re.fullmatch(self.identif, word)
            if word not in self.idents.keys():
                self.idents[word] = self.id_idents
            self.id_idents += 1
            self.table.append(word + 'Идентификатор;;' + str(self.idents[word]))
        except AssertionError:
            self.handle(word)
        word = text.pop(0)
        try:
            assert word in self.compare
            self.table.append(word + 'Оператор; сравнения')
        except AssertionError:
            print('Invalid symbol: '+word+', comparison expected')
        word = text.pop(0)
        try:
            assert re.fullmatch(self.identif, word) or re.fullmatch(self.number, word)
            if re.fullmatch(self.identif, word):
                if word not in self.idents.keys():
                    self.idents[word] = self.id_idents
                self.id_idents += 1
                self.table.append(word + 'Идентификатор;;' + str(self.idents[word]))
            else:
                self.table.append(word + 'Вещественная; константа;' + word)
        except AssertionError:
            self.handle(word)
        word = text.pop(0)
        try:
            assert word == 'then'
            if word not in self.kws.keys():
                self.kws[word] = self.id_kws
            self.id_kws += 1
            self.table.append(word + 'Ключевое; словоX;' + str(self.kws[word]))
        except AssertionError:
            print('Invalid keyword: '+word+', `then` expected')
        extract_base()
        if text[0] != 'else':
            return text
        else:
            word = text.pop(0)
            if word not in self.kws.keys():
                self.kws[word] = self.id_kws
            self.id_kws += 1
            self.table.append(word + 'Ключевое; словоX;' + str(self.kws[word]))
            extract_base()
        return text
```

Метод remove_comments() нацелен на удаление комментариев из входного текста. Поскольку условие задания допускало выбор любой формы комментариев, в данной реализации был выбран символ '#' как начало комментария и перенос строки ('\n') как его конец. Этот метод удаляет из текста, поступившего на вход, все лексемы, находящиеся между указанными символами, включая их самих.

Вход: text - список символов, из которого необходимо удалить комментарии

Выход: метод не возвращает никакого значения

```
def remove_comments(self, text):
    indexes = []
    for i in range(len(text)):
        if '#' in text[i]:
            indexes.append(i)
    numb_pop = 0
    for i in indexes:
        text.pop(i - numb_pop)
        numb_pop += 1
    for line in text:
        for item in line.split():
            self.text.append(item)
```

Метод handle() определяет тип ошибки, к которому относится поступивший на вход некорректный идентификатор, и выводит текст этой ошибки в консоль.
Вход: word - строка, содержащая некоторую ошибку
Выход: метод не возвращает никакого значения

```
def handle(self, word):
    if len(word) >= 16:
        print('Invalid identifier: '+word+', line too long')
    elif re.match(r'[0-9]', word):
        print('Invalid identifier: '+word+', line contains invalid symbols')
```

2.2 IO.py

Класс IO_worker реализует работу с файлом, путь к которому указывает пользователь. В случае, если такой файл существует, метод read() возвращает его содержимое. В противном случае этот метод выводит сообщение об ошибке и возвращает значение None.
Вход: infilepath - строка, содержащая путь к файлу
Выход: метод не возвращает никакого значения

```
class IO_worker:
    def __init__(self, infilepath):
        self.infilepath = infilepath

    def read(self):
        try:
            infile = open(self.infilepath, 'r')
            return infile.read()
        except FileNotFoundError:
            print("No such file! Try again.")
            return None
```

2.3 main.py

Файл main.py содержит основной цикл программы, в котором реализуется взаимодействие с пользователем через консольный ввод-вывод, а также создаются объекты классов IO и Analyser, из которых вызываются методы, организующие работу с входным файлом. Кроме того, в этом файле содержится сохранение таблицы-результата в формате csv посредством библиотеки pandas.

```
from IO import IO_worker
from Analyser import Analyser
import pandas

while True:
    print('Enter text path: ')
    text = IO_worker(input()).read()
    if text:
        analyser = Analyser(text)
        analyser.analyse()
        print('No errors detected!')
        table = []
        for row in analyser.get_table():
            table.append(row.split(';'))
        df = pandas.DataFrame(table, columns=['Лексема', 'Тип лексемы', 'Значение'])
        print('Enter output file path: ')
        df.to_csv(input(), index=False, encoding='utf-8')
        print('Data written to csv, would you like to also watch it here? y/n: ')
        if input() == 'y':
            print(df)
    print('Continue? y/n: ')
    if input() != 'y':
        break
```

3 Результаты работы программы

Ниже представлены примеры работы программы на различных входных данных. На рисунке 1 показана работа программы при корректных входных данных: существующий файл, который не содержит ошибок. На рисунке 2 изображена реакция программы на ввод имени несуществующего файла. На рисунке 3 представлен результат обработки входного файла, содержащего лексические ошибки. На рисунке 4 показан фрагмент таблицы лексем, сгенерированной программно в формате csv.

```
Enter text path:
files/file1.txt
No errors detected!
Enter output file path:
files/out.csv
Data written to csv, would you like to also watch it here? y/n:
y
     Лексема             Тип лексемы  Значение
0           a           Идентификатор         1
1          :=       Знак присваивания      None
2         0.5  Вещественная константа       0.5
3           b           Идентификатор         2
4          :=       Знак присваивания      None
5           a           Идентификатор         1
6          if          Ключевое слово        X1
7           a           Идентификатор         1
8           =      Оператор сравнения      None
9           b           Идентификатор         2
10       then          Ключевое слово        X2
11          c           Идентификатор         6
12         :=       Знак присваивания      None
13        2.7  Вещественная константа       2.7
```

Рис. 1: Работа программы с корректными входными данными

```
Enter text path:
files/nosuchfile.txt
No such file! Try again.
Continue? y/n:
```

Рис. 2: Реакция программы на несуществующий входной файл

```
Enter text path:
files/file1.txt
Invalid identifier: bbbbbbbbbbbbbbbbbbbbbbbbbbbb, line too long
Invalid identifier: 2.7l, line contains invalid symbols
Invalid symbol: ?, `:=` expected
```

Рис. 3: Реакция программы на ошибки во входном файле

Лексема	Тип лексемы	Значение
a	Идентификатор	1
:=	Знак присваивания	
0.5	Вещественная конста	0.5
b	Идентификатор	2
:=	Знак присваивания	
a	Идентификатор	1
if	Ключевое слово	X1
a	Идентификатор	1
=	Оператор сравнения	
b	Идентификатор	2
then	Ключевое слово	X2
c	Идентификатор	6
:=	Знак присваивания	
2.7	Вещественная конста	2.7
a	Идентификатор	1
:=	Знак присваивания	
c	Идентификатор	6
else	Ключевое слово	X3
c	Идентификатор	6
:=	Знак присваивания	
3.1	Вещественная конста	3.1
a	Идентификатор	1
:=	Знак присваивания	
2.2	Вещественная конста	2.2
b	Идентификатор	2
:=	Знак присваивания	
1.38E-23	Вещественная конста	1.38E-23
if	Ключевое слово	X1
a	Идентификатор	1
>	Оператор сравнения	
0	Вещественная конста	0

Рис. 4: Сгенерированная таблица лексем в формате csv

Заключение

В результате выполнения данной лабораторной работы была создана программа на языке Python, реализующая лексический анализатор. Программа считывает входной текст, анализирует его на соответствие правилам заданного языка и генерирует таблицу лексем, а также выводит сообщения о встреченных ошибках. Предусмотрен пользовательский ввод через консоль и считывание данных из файла.

Достоинства программы

1. Обработка некорректного пользовательского ввода.
2. Вывод таблицы лексем как в консоль, так и в файл формата csv.
3. Использование рекурсивных методов, упрощающее понимание программного кода пользователем.
4. Использование регулярных выражений для проверки корректности идентификаторов.

Недостатки программы

1. Отсутствие меню, позволяющего выбрать необходимую операцию.
2. Узкая специализация - при изменении входного языка потребуется редактирование значительной части кода.

Список литературы

[1] Теория автоматов Ю.Г. Карпов - СПб.: Питер, 2003. - 208 с.

[2] Курс лекций по математической логике и теории автоматов. Востров А.В. https://tema.spbstu.ru/mathem/ (Дата последнего обращения: 04.06.2022)