INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO (ESCOM)

TEORIA COMPUTACIONAL

NOMBRE Y NÚMERO DE LA PRÁCTICA:

- PRÁCTICA 7. GRAMÁTICAS

NOMBRE DEL ALUMNO:

- SANTOS MÉNDEZ ULISES JESÚS

NOMBRE DEL MAESTRO:

- JORGE LUIS ROSAS TRIGUEROS

FECHA DE REALIZACIÓN:

- 08/01/2021

FECHA DE ENTREGA:

- 15/01/2021

Marco Teórico:

Lenguajes y gramáticas independientes del contexto

Los lenguajes independientes del contexto, son lenguajes que utilizan una notación natural recursiva, estas gramáticas han desarrollado un importante papel en la tecnología de compiladores.

Se han hecho que la implementación de analizadores sintácticos (funciones que descubren la estructura de un programa), las gramáticas independientes del contexto se han utilizado para describir formatos de documentos a través de la denominada definición de tipo de documento.

PLY una Implementación de lex & yacc en Python

PLY es una implementación de lex y yacc de herramientas de análisis para Python.

Sus características:

- PLY proporciona la mayor parte de las características estándar del Lex / Yacc incluido el apoyo a las producciones vacías, las reglas de prioridad, la recuperación de errores y soporte para gramáticas ambiguas.

- PLY consta de dos módulos separados; lex.py y yacc.py, ambos de los cuales se encuentran en un paquete de Python llamado capa. El módulo lex.py se utiliza para romper texto de entrada en una colección de ficha especificadas por una colección de reglas de expresiones regulares yacc.py se utiliza para reconocer la sintaxis del lenguaje que se ha especificado en el formulario de una gramática libre de contexto. yacc.py utiliza análisis sintáctico LR y genera sus tablas de análisis sintáctico utilizando el LALR (1) (por defecto) o algoritmos de generación de tabla SLR .

- Las dos herramientas tienen el propósito de trabajar juntos. Específicamente, lex.py ofrece una interfaz externa en forma de una función de señal () que devuelve el siguiente token válido en el flujo de entrada. yacc.py llama a esto varias veces para recuperar tokens e invocar las reglas gramaticales. La salida de yacc.py es a menudo un árbol de sintaxis abstracta (AST). Sin embargo, esto es totalmente en manos del usuario. Si se desea, yacc.py también se puede utilizar para implementar simples compiladores de una pasada.

- PLY consta de dos archivos: lex.py y yacc.py. Estos están contenidos dentro del directorio ‘ply’ que también puede ser utilizado como un paquete de Python. Para utilizar capas, simplemente copia el directorio ‘ply’ de su proyecto y de importación lex y yacc del paquete de ‘ply’ asociado.

Material y Equipo:

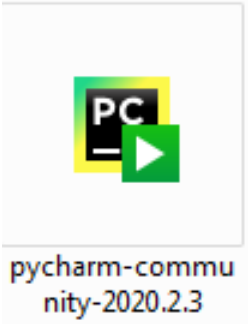
-PC (véase figura 1)[1]

(Figura 1)

-Python 3 (Python 3.8.6) (véase figura. 2)[2]

(Figura 2)

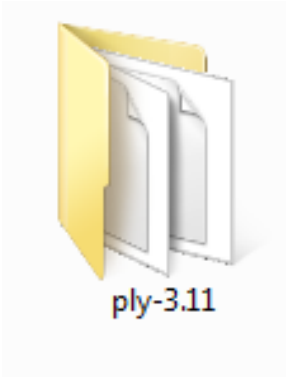
-IDE (Pycharm) (véase figura 3)[3]



(Figura 3)

[1] Figura 1: PC o cualquier computadora para el desarrollo de la práctica.
[2] Ejecutable para instalar Python 3.8.6.
[3] Ejecutable de IDE Pycharm .

-PLY lex & yacc para Python (véase figura 4)[4]



(Figura 4)

[4] Figura de la carpeta donde se encuentra la subcarpeta PLY.

Desarrollo:

1) Primero se mostró un ejemplo del funcionamiento de PLY haciendo las diversas producciones para permitir la entrada del lenguaje a partir del teclado, primero se parte por definir los token y las expresiones regulares que serán empleadas, después se hacen diversas funciones donde una de ellas se encarga de recibir como parámetro a t que es la variable a la que se le asigna el resultado de compilar "parsear" el código y el resultado se compara con el analizador léxico para ver si es un carácter que esté dentro del lenguaje, también realiza producciones que se comparan para ver si pertenecen o no al lenguaje. (Véase Figura. 5)[5]

```
# a^n b^n
tokens = ('a', 'b');
t_a = r'a';
t_b = r'b';

def t_error(t):
    print("Caracter ilegal ", t.value[0])
    t.lexer.skip(1)

import ply.lex as lex
lex.lex()

def p_S(p):
    ''' S : a S b
          | empty'''
    pass

def p_empty(p):
    'empty :'
    pass

s = '';

def p_error(p):
    global s
    if p:
        print("Error de sintaxis en ", p.value)
    else:
        print("Error de sintaxis en EOF")
    print(s, "no está en el lenguaje")

import ply.yacc as yacc
yacc.yacc()

while 1:
    try:
        s = input('> ')
    except EOFError:
        break
    t = yacc.parse(s)
```

(Figura 5)

[5] Código del programa que reconoce mismo número de a's que de b's.

2) La ejecución del programa se logra observar de la siguiente manera (Véase Figura. 6)[6]

```
Generating LALR tables
> aabb
> aaabbb
>
> a
Error de sintaxis en EOF
a no está en el lenguaje
> b
Error de sintaxis en  b
b no está en el lenguaje
> c
Caracter ilegal  c
> teoria
Caracter ilegal  t
Caracter ilegal  e
Caracter ilegal  o
Caracter ilegal  r
Caracter ilegal  i
Error de sintaxis en EOF
teoria no está en el lenguaje
> aaaaaabbbbb
Error de sintaxis en EOF
aaaaaabbbbb no está en el lenguaje
```

(Figura 6)

3) Se dejaron 3 problemas, el primero consiste en hacer un programa que reconozca $\{w\epsilon\{a,b,c\}^*|w\ no\ tiene\ la\ subcadena\ ac\}$, primero se hace el planteamiento de los tokens (Véase Figura. 7)[7]

```
2   print("Programa que reconoce {wÉ{a,b,c}*|w no tiene la subcadena  ac}")
3   tokens = ('a', 'b', 'c');
4   t_a = r'a';
5   t_b = r'b';
6   t_c = r'c';
```

(Figura 7)

[6] Datos y resultados en la ejecución del código.
[7] Codificación de tokens y de expresiones regulares de a, b y c.

4) Después hacemos una función que reciba como parámetro el valor de t la variable donde se guarda el resultado hecho por el compilador, después importamos en analizador léxico (Véase Figura. 8)[8]

```
8   # Definimos la funcion de error por si hay algun caracter no valido dentro del lenguaje
9   def t_error(t):
10      print("Caracter ilegal ", t.value[0])
11      t.lexer.skip(1)
12
13  import ply.lex as lex
14  lex.lex()
```

(Figura 8)

5) Hacemos las distintas producciones con su no terminal S y el conjunto de terminales que harán las debidas producciones (Véase Figura. 9)[9]

```
16  # funcion encargada de las producciones de la forma de Backus Naur
17  def p_S(p):
18      ''' S : a X
19            | b S
20            | c S
21            | empty'''
22      pass
23
24  # funcion que permita la produccion de epsilon
25  def p_empty(p):
26      'empty :'
27      pass
28
29  # definimos la produccion de X
30  def p_X(p):
31      ''' X : b X
32            | empty'''
33      pass
34
35  s = '';
```

(Figura 9)

[8] Carga y análisis de resultado del analizador lex.py.
[9] Zona de código en la que se hacen distintas producciones cada una en una función.

6) Se hizo una función para analizar a p y determinar si pertenece o no al lenguaje, al final se encuentra el compilador que recibe como parámetro a s que es el encargado de procesar lo que se ingrese por teclado de manera que el resultado se le asignará a t y así se determinará si es correcto o no. (Véase Figura. 10)[10]

```
37  # Funcion para la impresion de errores
38  def p_error(p):
39      global s
40      if p:
41          print("Error de sintaxis en ", p.value)
42      else:
43          print("Error de sintaxis en EOF")
44
45      print(s, "no está en el lenguaje")
46
47  import ply.yacc as yacc
48  yacc.yacc()
49  while 1:
50      try:
51          s = input('> ')
52      except EOFError:
53          break
54      t = yacc.parse(s)
```

(Figura 10)

7) La ejecución del programa se logra observar de la siguiente manera (Véase Figura. 11)[11]

```
Programa que reconoce {wÉ{a,b,c}*|w no tiene la subcadena  ac}
>
> a
> b
> c
> ab
> bc
> ac
Error de sintaxis en  c
ac no está en el lenguaje
> tgk
Caracter ilegal  t
Caracter ilegal  g
Caracter ilegal  k
```

(Figura 11)

[10] Zona de código donde se genera el análisis de producciones y se determina si pertenece o no al lenguaje.
[11] Ejecución del con las distintas combinaciones pertenecientes y no pertenecientes al lenguaje.

8) Se dejaron 3 problemas, el segundo consiste en hacer un programa que reconozca $\{a^{(n+3)}b^n\}$, primero se hace el planteamiento de los tokens (Véase Figura. 12)[12]

```
print("Programa que reconoce a^(n+3)b^n")
tokens = ('a', 'b');
t_a = r'a';
t_b = r'b';
```

(Figura 12)

9) Después hacemos una función que reciba como parámetro el valor de t la variable donde se guarda el resultado hecho por el compilador, después importamos en analizador léxico (Véase Figura. 13)[13]

```
# Definimos la funcion de error por si hay algun caracter no valido dentro del lenguaje
def t_error(t):
    print("Caracter ilegal ", t.value[0])
    t.lexer.skip(1)

import ply.lex as lex
lex.lex()
```

(Figura 13)

[12] Formación de tokens y expresiones regulares para a y b.
[13] Función que reconoce caracteres no válidos en el lenguaje.

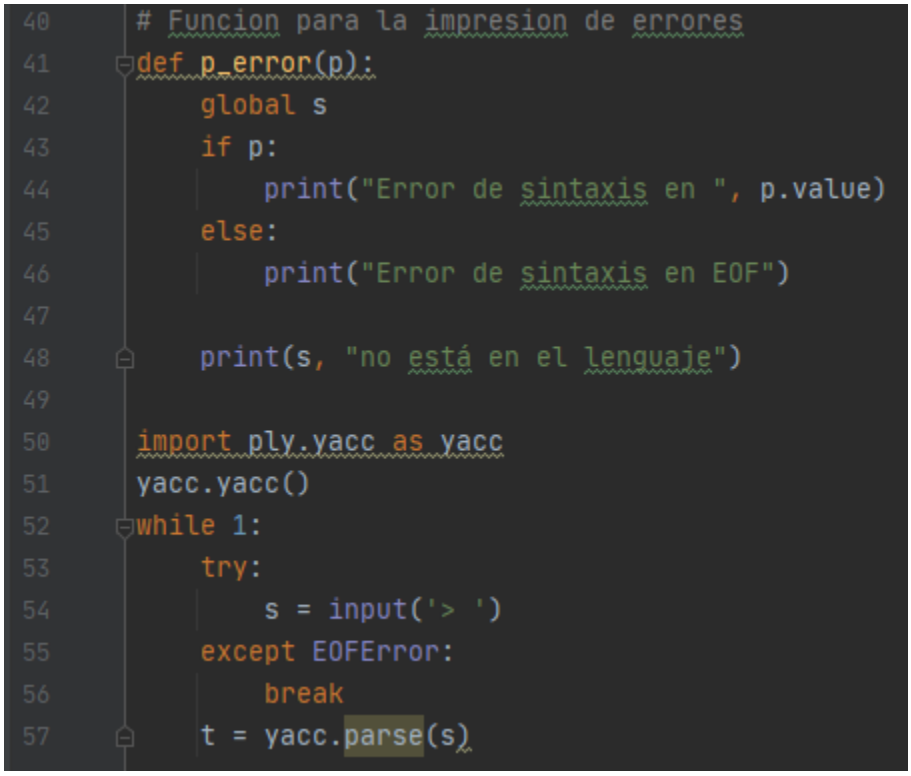
10) Hacemos las distintas producciones con su no terminal S y el conjunto de terminales que harán las debidas producciones (Véase Figura. 14)[14]

```
14    # funcion encargada de las producciones de la forma de Backus Naur
15    def p_S(p):
16        ''' S : a a a
17              | a a a a b
18              | a a a a X b b T'''
19        pass
20
21    # funcion que permita la produccion de epsilon
22    def p_empty(p):
23        'empty :'
24        pass
25
26    # definimos la produccion de X
27    def p_X(p):
28        ''' X : a X
29              | empty'''
30        pass
31
32    # definimos la produccion de T
33    def p_T(p):
34        ''' T : b T
35              | empty'''
36        pass
37
38    s = '';
```

(Figura 14)

[14] Diversas producciones para generar el lenguaje.

11) Se hizo una función para analizar a p y determinar si pertenece o no al lenguaje, al final se encuentra el compilador que recibe como parámetro a s que es el encargado de procesar lo que se ingrese por teclado de manera que el resultado se le asignará a t y así se determinará si es correcto o no. (Véase Figura. 15)[15]



```
# Funcion para la impresion de errores
def p_error(p):
    global s
    if p:
        print("Error de sintaxis en ", p.value)
    else:
        print("Error de sintaxis en EOF")

    print(s, "no está en el lenguaje")

import ply.yacc as yacc
yacc.yacc()
while 1:
    try:
        s = input('> ')
    except EOFError:
        break
    t = yacc.parse(s)
```

(Figura 15)

[15] Función que recibe a p de las funciones y determina si pertenece o no al lenguaje que se está proponiendo.

12) La ejecución del programa se logra observar de la siguiente manera (Véase Figura. 16)[16]

```
Generating LALR tables
WARNING: 1 shift/reduce conflict
> aaa
>
Error de sintaxis en EOF
 no está en el lenguaje
> ab
Error de sintaxis en  b
ab no está en el lenguaje
> aaaab
> aaaaabb
> ffjk
Caracter ilegal  f
Caracter ilegal  f
Caracter ilegal  j
Caracter ilegal  k
Error de sintaxis en EOF
ffjk no está en el lenguaje
```

(Figura 16)

13) Se dejaron 3 problemas, el tercero consiste en hacer un programa que reconozca $\{a^i\ b^j\ c^k \mid i = j\ o\ j = k\}$, primero se hace el planteamiento de los tokens (Véase Figura. 17)[17]

```
1  print("Programa que reconoce {a^i b^j c^k | i =j o j = k}")
2  tokens = ('a', 'b', 'c');
3  t_a = r'a';
4  t_b = r'b';
5  t_c = r'c';
```

(Figura 17)

[16] Ejecución del programa donde se ingresaron distintas cadenas válidas y no válidas.
[17] Propuesta de tokens y expresiones regulares a analizar.

14) Después hacemos una función que reciba como parámetro el valor de t la variable donde se guarda el resultado hecho por el compilador, después importamos en analizador léxico (Véase Figura. 18)[18]

```
7   # Definimos la funcion de error por si hay algun caracter no valido dentro del lenguaje
8   def t_error(t):
9       print("Caracter ilegal ", t.value[0])
10      t.lexer.skip(1)
11
12  import ply.lex as lex
13  lex.lex()
```

(Figura 18)

[18] Función que determina si hay caracteres ilegales en la cadena.

15) Hacemos las distintas producciones con su no terminal S y el conjunto de terminales que harán las debidas producciones (Véase Figura. 19)[19]

```
15      # funcion encargada de las producciones de la forma de Backus Naur
16      def p_S(p):
17          ''' S : X
18                | Y'''
19          pass
20
21      # funcion que permita la produccion de epsilon
22      def p_empty(p):
23          'empty :'
24          pass
25
26      # definimos la produccion de X
27      def p_X(p):
28          ''' X : Z T'''
29          pass
30
31      # definimos la produccion de T
32      def p_Z(p):
33          ''' Z : a Z b
34                | empty'''
35          pass
36
37      def p_T(p):
38          ''' T : c T
39                | empty'''
40          pass
41
42      def p_Y(p):
43          ''' Y : F W'''
44          pass
45
46      def p_W(p):
47          ''' W : b W c
48                | empty'''
49          pass
50
51      def p_F(p):
52          ''' F : a F
53                | empty'''
54          pass
```

(Figura 19)

[19] Zona donde se encuentran las distintas producciones para generar el lenguaje.

16) Se hizo una función para analizar a p y determinar si pertenece o no al lenguaje, al final se encuentra el compilador que recibe como parámetro a s que es el encargado de procesar lo que se ingrese por teclado de manera que el resultado se le asignará a t y así se determinará si es correcto o no. (Véase Figura. 20)[20]

```
s = '';

# Funcion para la impresion de errores
def p_error(p):
    global s
    if p:
        print("Error de sintaxis en ", p.value)
    else:
        print("Error de sintaxis en EOF")

    print(s, "no está en el lenguaje")

import ply.yacc as yacc
yacc.yacc()
while 1:
    try:
        s = input('> ')
    except EOFError:
        break
    t = yacc.parse(s)
```

(Figura 20)

[20] Función que determina si el parámetro p pertenece o no al lenguaje.

17) La ejecución del programa se logra observar de la siguiente manera (Véase Figura. 21)[21]

```
Programa que reconoce {a^i b^j c^k | i =j o j = k}
Generating LALR tables
WARNING: 2 reduce/reduce conflicts
WARNING: reduce/reduce conflict in state 7 resolved using rule (Z -> empty)
WARNING: rejected rule (F -> empty) in state 7
WARNING: Rule (F -> empty) is never reduced
>
> abc
> ab
> abccccc
> a
Error de sintaxis en EOF
a no está en el lenguaje
> ca
Error de sintaxis en  a
ca no está en el lenguaje
> uli
Caracter ilegal  u
Caracter ilegal  l
Caracter ilegal  i
```

(Figura 21)

[21] Ejecución del programa con distintas cadenas válidas y no válidas.

Conclusiones:

En conclusión esta práctica nos enseñó a realizar distintas producciones para generar una gramática, mostrándonos unas de las aplicaciones de analizadores léxicos sintácticos para la solución de un problema y además se dio una introducción breve a los compiladores.

Bibliografía:

- *Python para todos*, Raúl González Duque, 2007.
- *Introducción a la programación en Python*, Andrés Marzal, Isabel García, 2010.
- *Teoría de autómatas y lenguajes formales*, Serafín Moral.
- *Teorías de Autómatas y lenguajes formales*, Elena Jurado Málaga, 2008.
- *Teoría de la computación; Lenguajes formales, autómatas y complejidad,* J. Glenn Brookshear.
- *Teoría de autómatas, lenguajes y computación,* Jeffrey Ullman, 2007.