INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

MATERIA: Teoría Computacional.

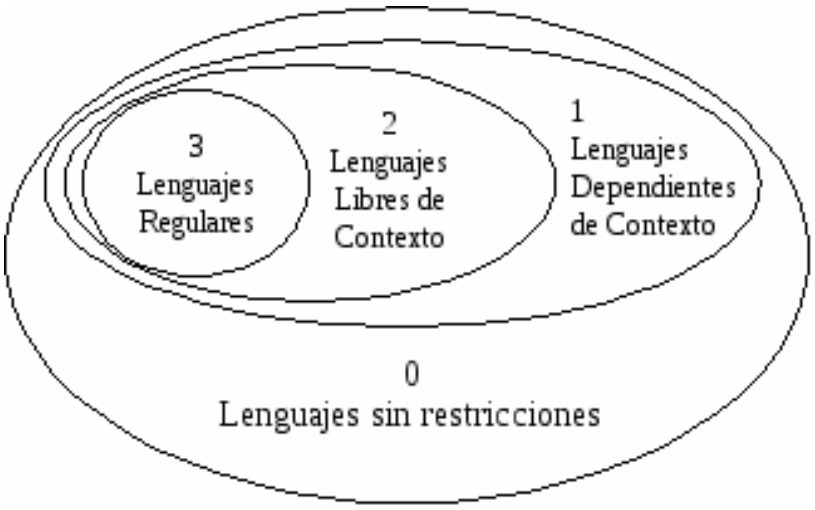
PRÁCTICA: Práctica 7. Gramáticas.

ALUMNO: Meza Vargas Brandon David.

PROFESOR: Jorge Luis Rosas Trigueros.

FECHA PRÁCTICA: 08-ene-2021

FECHA DE ENTREGA: 15-ene-2021



MARCO TEÓRICO

Gramática Regular

Una gramática regular G es una 4-tupla G=(Σ, N, S, P), donde

Σ - es un alfabeto,

N - es una colección de símbolos no terminales,

S - es un no terminal llamado símbolo inicial, y

P - es una colección de reglas de reescritura llamadas también producciones.

En una gramática regular, las producciones son de la forma

A → w, donde A ϵN y w es una cadena sobre (ΣUN) * que satisface lo siguiente:

1.- w contiene un no terminal como máximo.

2.- Si w contiene un no terminal, entonces es el símbolo que está en el extremo derecho de w.

El lenguaje generado por la gramática regular G se denota por L(G).

La definición que manejamos de Gramática Regular hace que las cadenas sean generadas de izquierda a derecha. Por esta razón, este tipo de gramática puede llamarse gramática regular por la derecha. Una gramática regular por la izquierda es aquella cuyas cadenas son generadas de derecha a izquierda, es decir, las producciones son pares de

Nx(NUε)Σ*.

Ejemplo:

Obtener una gramática regular por la izquierda para el lenguaje { anbaa | n≥0 }.

S→ Abaa

A→ Aa | ε

Derivar baa

S⇒ Abaa ⇒ εbaa ⇒ baa

Derivar aaabaa

S⇒ Abaa ⇒ Aabaa ⇒ Aaabaa ⇒ Aaaabaa ⇒ εaaabaa ⇒ aaabaa

Obtener una gramática regular por la derecha para el lenguaje { anbaa | n≥0 }.

$S \rightarrow aS \mid baa$

Derivar baa

$S \Rightarrow baa$

Derivar aaabaa

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaabaa$

Gramáticas Independientes del Contexto

Una gramática independiente del contexto (GIC) es una 4-tupla

$G=(N, \Sigma, S, P)$

donde N es una colección finita de no terminales, Σ es un alfabeto (conjunto de terminales), S es un no terminal determinado que se llama símbolo inicial y $P \subseteq N \times (N \cup \Sigma)^*$ es un conjunto de producciones.

Ejemplo:

$S \rightarrow aSb \mid \varepsilon$

$S \Rightarrow \varepsilon$

$S \Rightarrow aSb \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow a\ aSb\ b \Rightarrow aa(\varepsilon)bb \Rightarrow aabb$

$L(G)=\{ a^n b^n \}$

El lenguaje generado por la GIC G se denota por L(G) y se llama lenguaje independiente del contexto (LIC).

PLY (Python Lex-Yacc)

En pocas palabras, PLY es nada más que una aplicación de lex / yacc. Aquí está una lista de sus características esenciales:

- Se implementa completamente en Python.
- Se utiliza el análisis sintáctico LR – que es razonablemente eficiente y muy adecuado para gramáticas más grandes.
- PLY proporciona la mayor parte de las características estándar de lex / yacc incluido el apoyo a las producciones vacías, las reglas de prioridad, la recuperación de errores y soporte para gramáticas ambiguas.
- PLY es sencillo de usar y proporciona muy extensa comprobación de errores.
- PLY no trata de hacer nada más ni menos que proporcionar la funcionalidad básica lex / yacc. En otras palabras, no es un marco de análisis sintáctico grande o un componente de algún sistema más grande.

PLY consta de dos módulos separados; lex.py y yacc.py, ambos de los cuales se encuentran en un paquete de Python llamada capa. El módulo lex.py se utiliza para romper texto de entrada en una colección de fichas 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.

MATERIAL Y EQUIPO

Para la realización de esta práctica son necesarias las siguientes herramientas:

- Un equipo de cómputo que cumpla con los requerimientos para el uso del lenguaje de programación Python.
- Tener instalado el lenguaje de programación Python.
- Tener la librería PLY (Python Lex-Yacc) para trabajar con gramáticas.
- Contar con un IDE para programar con Python, cualquiera es útil

DESARROLLO

El desarrollo de esta práctica consistió en codificar algunas gramáticas, para ser exactos 4, esto gracias a la librería PLY.

Primeramente, se descargó la librería PLY desde su sitio oficial el cuál es el siguiente https://www.dabeaz.com/ply/, una vez descargada se arrastró al directorio en donde se trabajaron los programas con las gramáticas, antes de empezar con los ejercicios propuestos, se descargó el ejemplo que viene en la página anteriormente mostrada; por lo tanto, en la figura 1 vemos cómo funciona este ejemplo, el cual es una calculadora:

```
calc > 5+4*6
29
calc > 3-3*213
-636
calc > (3-3)*3213
0
calc >
```

Figura 1. Ejemplo de PLY

Para empezar con las gramáticas vimos un ejemplo que proporciono el profesor, fue el lenguaje a^n b^n, en esta parte se explicará el código para ya no explicarlo en los lenguajes posteriores, pues lo único que cambia es la gramática que genera el lenguaje.

En la figura 2 tenemos la primera parte, en donde tenemos los símbolos que conforman el alfabeto, así como una función que detecta cuando ingresamos un símbolo que no pertenece al alfabeto;

```
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)
```

Figura 2. Alfabeto y función para detectar caracteres ilegales.

Posteriormente, importamos ply.lex, después tenemos dos funciones más, una es la gramática, en esta ocasión, es la gramática que genera a^n b^n, la segunda función es para aceptar la cadena vacía, aquí esta comentada ya que este lenguaje no genera esta cadena, ver figura 3.

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

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

##def p_empty(p):

##    'empty :'

##    pass
```

Figura 3. Importación de ply.lex, función con la gramática y cadena vacía.

La función mostrada en la figura 4, se encarga de ver si una cadena no está en el lenguaje, además de decir si esta tiene un error de sintaxis;

```
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")
```

Figura 4. Función que ve errores en la sintaxis y si una cadena no está en el lenguaje.

Por último, importamos ply.yacc para generar el analizador sintáctico, además tenemos un ciclo infinito donde se almacenara en s lo que vayamos escribiendo y en la línea t= yacc.parse(s) se parsea lo que introdujimos con el teclado, esto lo vemos en la figura 5;

```
import ply.yacc as yacc
yacc.yacc()

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

Figura 5. Parte final del código.

Ahora veamos cómo funciona este programa probando con algunas cadenas, ver figura 6;

```
> aabb
> aaabbb
> aaaaabbbbb
> ababa
Error de sintaxis en  a
ababa no está en el lenguaje
> baba
Error de sintaxis en  b
baba no está en el lenguaje
> bbaa
Error de sintaxis en  b
bbaa no está en el lenguaje
> aaab
Error de sintaxis en EOF
aaab no está en el lenguaje
>
```

Figura 6. Prueba del lenguaje.

Ahora sí, pasemos a los ejercicios

Problema 1

El primero es generar la gramática para el lenguaje: {w ϵ {a, b} *| w termina en b} y pasarla a código.

La gramática que genera el lenguaje anterior es la siguiente:

S → a S | b S | b

Una vez obtenida la gramática lo pasamos al código, el código lo vemos en la figura 7;

```
# { w ϵ { a, b}* | w termina en b }
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 S
          | b'''
    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 7. Programa para el problema 1.

Ahora veamos cómo funciona este programa probando con algunas cadenas, ver figura 8;

```
Generating LALR tables
> a
Error de sintaxis en EOF
a no está en el lenguaje
> abbba
Error de sintaxis en EOF
abbba no está en el lenguaje
> b
> bbbb
> aaab
> ababbbb
> ababb
> aaabb
> c
Caracter ilegal  c
Error de sintaxis en EOF
c no está en el lenguaje
>
```

Figura 8. Prueba del lenguaje.

Problema 2

El segundo es generar la gramática para el lenguaje {w ϵ {a,b,c}*| w no tiene la subcadena ac} y pasarla a código

La gramática que genera el lenguaje anterior es la siguiente:

S → S a | S b S | c X | ε

X → c X | S

El código lo podemos ver en la figura 9, como se mencionó anteriormente, no se explicará el código, pues este ya fue explicado, solo se implementó la gramática de este problema y se agregó c al alfabeto.

```
#{w ϵ {a,b,c}*| w no tiene la subcadena ac}
tokens = ('a', 'b', 'c');
t_a = r'a';
t_b = r'b';
t_c = r'c';


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 : S a
          | S b S
          | c X
          | empty
        X : c X
          | S'''
    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 9. Programa para el problema 2.

Ahora veamos cómo funciona este programa probando con algunas cadenas, ver figura 10;

```
> abbbbbbbababababccaaa
> ccacaca
Error de sintaxis en  c
ccacaca no está en el lenguaje
> abcabcabc
> bcbcbcbcababababab
> abbbbababcabcbca
> abbbbbbccccaaabbbababbbac
Error de sintaxis en  c
abbbbbbccccaaabbbababbbac no está en el lenguaje
> ac
Error de sintaxis en  c
ac no está en el lenguaje
>
> bbac
Error de sintaxis en  c
bbac no está en el lenguaje
> abcabcabcabcabc
>
```

Figura 10. Prueba del lenguaje.

Problema 3

El tercero es generar la gramática para el lenguaje {a^(n+3)b^n} y pasarla a código.

La gramática que genera el lenguaje anterior es la siguiente:

S → aaaT

T → aTb | ε

El código lo podemos ver en la figura 11;

```
# { a^(n+3)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 a a T
        T : a T 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 11. Programa para el problema 3.

Ahora veamos cómo funciona este programa probando con algunas cadenas, ver figura 12;

```
Generating LALR tables
>
Error de sintaxis en EOF
 no está en el lenguaje
> aaa
> b
Error de sintaxis en  b
b no está en el lenguaje
> babab
Error de sintaxis en  b
babab no está en el lenguaje
> aaab
Error de sintaxis en  b
aaab no está en el lenguaje
> aaaab
> aaaaabb
> aaaaaabbb
> aaaaaaabbbb
> aaabb
Error de sintaxis en  b
aaabb no está en el lenguaje
>
```

Figura 12. Prueba del lenguaje.

Problema 4

El cuarto es generar la gramática para el lenguaje {a^i b^j c^k | i=j ó j=k} y pasarla a código.

La gramática que genera el lenguaje anterior es la siguiente:

S → X | Y
X→ ZT
Z→ aZb | ε # i=j
T→ cT | ε

Y→ FW
W→ bWc | ε # j=k
F→ aF | ε

El código lo podemos ver en la figura 13;

```
#{ a^i b^j c^k | i=j ó j=k }
tokens = ('a', 'b', 'c');
t_a = r'a';
t_b = r'b';
t_c = r'c'

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 : X
          | Y
        X : Z T
        Z : a Z b
          | empty
        T : c T
          | empty
        Y : F W
        F : a F
          | empty
        W : b W c
          | 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 13. Programa del problema 4

Ahora veamos cómo funciona este programa probando con algunas cadenas, ver figura 14;

```
> abc
> ab
> abccccc
> abbbb
Error de sintaxis en  b
abbbb no está en el lenguaje
> abbbc
Error de sintaxis en  b
abbbc no está en el lenguaje
>
>
```

Figura 14. Prueba del lenguaje.

CONCLUSIONES Y RECOMENDACIONES

A partir de los ejercicios realizados en la práctica, pude comprender de una mejor manera lo revisado en clases anteriores sobre gramáticas, de igual manera me sirvió como repaso ya que estos temas fueron vistos antes de vacaciones, por lo que siento que fue un momento adecuado para hacer la práctica.

Tengo que decir que los ejercicios fueron sencillos de resolver, pero si tuve algunos problemas con el problema 3 para formar su gramática, pero al final lo logre de una buena forma.

La práctica fue realizada de manera clara y adecuada, pues el profesor resolvió dudas mientras realizábamos la práctica, además, como mencione anteriormente, fue hecha en un tiempo adecuado ya que recién regresamos de vacaciones y realizar esta práctica ayudo a recordar.

BIBLIOGRAFÍA

- Apuntes de la clase de Teoría Computacional por el profesor Jorge Luis Rosas Trigueros.
- dabez. “PLY (Python Lex-Yacc). Obtenido de: https://www.dabeaz.com/ply/”
- Zhao X. (2019). “gramáticas regulares”. Obtenido de: https://www.geeksforgeeks.org/regular-expressions-regular-grammar-and-regular-languages/
- Cueva, J. (2001). “Lenguajes Gramáticas y Autómatas”. Recuperado de: http://www.reflection.uniovi.es/ortin/publications/automata.pdf