Instituto Politécnico Nacional

Escuela Superior de Computación

Teoría de la Computación

Práctica 7

Gramática No Ambigua

Profesor: Juarez Martinez Genaro

Alumno: Jimenez Luna Rodrigo Efren

Email: jimenez.luna.efren@gmail.com

Grupo: 4CM2

Índice

1. Introducción

En el estudio de la teoría de la computación y el análisis gramatical, las gramáticas formales juegan un papel fundamental. Una gramática es una herramienta que nos permite describir y analizar la estructura de los lenguajes formales, como los lenguajes de programación o los lenguajes naturales.

Una característica deseable en una gramática es que sea no ambigua, es decir, que cada cadena del lenguaje pueda tener una única interpretación de acuerdo con las reglas de la gramática. En otras palabras, una gramática no ambigua es aquella en la cual no existe ninguna cadena que pueda ser generada por más de una derivación.

En esta práctica, exploraremos los conceptos de gramáticas no ambiguas y la implementación de una gramática que analiza pares de paréntesis

2. Marco Teórico

2.1. Gramática no ambigua

El análisis gramatical es una parte fundamental del estudio de la teoría de la computación y el lenguaje natural. Permite comprender y describir la estructura y las reglas de un lenguaje formal. Las gramáticas formales son herramientas utilizadas para modelar y analizar estos lenguajes, proporcionando reglas y producciones para generar y derivar cadenas válidas en el lenguaje [1].

Una gramática no ambigua es aquella en la cual cada cadena del lenguaje puede ser generada o derivada de una manera única y sin ambigüedad. Esto significa que no hay reglas de producción que permitan múltiples interpretaciones para una misma cadena [1].

La no ambigüedad en una gramática es un aspecto deseable debido a que facilita el análisis sintáctico de un lenguaje formal. Permite una interpretación inequívoca de las cadenas, evitando conflictos y ambigüedades en la comprensión de la estructura del lenguaje. Las gramáticas no ambiguas son más legibles, más fáciles de analizar y permiten la construcción de analizadores sintácticos eficientes [1].

3. Desarrollo

3.1. Programa 7 Gramática no Ambigua

En esta práctica se plante el siguiente problema: Implementar la gramática no ambigua que analiza cadenas de paréntesis ya sea ingresada por el usuario o de forma aleatoria e imprimir las descripciones instantáneas que va produciendo la gramática.

3.2. Planteamiento y solución

Para la solución de este problema se deberá implementar la gramática no ambigua que analice los pares de paréntesis, es decir que el uso de paréntesis sea sintácticamente correcto. En este caso es necesario utilizar producciones que no sean ambiguas. Las producciones para esta gramática son las siguientes:

- $B->(RB|\epsilon$
- $R->)|(RR$

Con esto, el código iniciará una nueva cadena con el símbolo inicial B e irá de forma recursiva reemplazando las variables por la producción a aplicar. Una vez llegado el final de la cadena de entrada, se analizará la cadena para saber si quedan variables en la cadena de producción. En caso de que si la cadena no es válida.

3.3. Código

A partir de la solución planteada en la sección anterior, se muestra el código implementado en el lenguaje de programación python, en este caso se separo en dos archivos diferentes, el primero dedicado para la función principal y el segundo para implementar los métodos:

3.3.1. Código de la función principal (main.py)

```
import os
import random

from functions import *

option = 1

while option:
    # Menu
    ##########################################################################
    print("Elija una de las siguientes opciones:")
    print("1) Modo Manual")
    print("2) Modo Aleatorio")
    print("3) Salir")

    option = int(input())

    os.system("clear")

    if option in (1, 2, 3):
        if option == 1:
```

```
            print("Ingrese una cadena de ('s y )'s menor de 1,000 caracteres: ")
            str = input()
            # print("{}".format(str))

            if validateStr(str):
                print("La cadena contiene caracteres diferentes de ( y )!\n")
                continue

            if len(str) > 1000:
                print("La cadena contiene m s caracteres de los permitidos\n")
                continue
        elif option == 2:
            str = createStr(random.randint(1, 100000))
            print("Evaluando la cadena de tama o {} en el automata...".format(l
        else:
            print("Sesion terminada")
            break;
    else:
        print("Opcion Invalida")
        continue;

    if CFG(str):
        print("La cadena es v lida!\n")
        print("Presione <Enter> para continuar\n")
        input()
        os.system("clear")
    else:
        print("La cadena no se encuentra en la gram tica!\n")
        print("Presione <Enter> para continuar\n")
        input()
        os.system("clear")
```

3.3.2. Código de los métodos utilizados (functions.py)

```
import random

def validateStr(str):
    for c in str:
        if not c in ("(", ")"):
            return 1

    return 0

def createStr(size):
    string = ""
```

```
    for i in range(size):
        string += random.choice(["(", ")"])

    return string

def CFG(str):
    output = open("output.txt", "w")
    steps = "B"
    str += " "

    output.write("Cadena: " + str + "\n\n")
    output.write("0, " + steps + "\n")

    for i, c in enumerate(str):
        if c == "(":
            if steps[i] == "B":
                steps = steps[0:i] + "(RB" + steps[i + 1:]
            elif steps[i] == "R":
                steps = steps[0:i] + "(RR" + steps[i + 1:]
        elif c == ")":
            if steps[i] == "B":
                break
            elif steps[i] == "R":
                steps = steps[0:i] + ")" + steps[i + 1:]
        elif c == " ":
            if steps[i] == "B":
                steps = steps[0:i] + "" + steps[i + 1:]

        output.write("{}, ".format(i + 1) + steps + "\n")

    output.close()

    if ("R" in steps) or ("B" in steps):
        return 0
    else:
        return 1
```

4. Resultados

Para demostrar el funcionamiento del programa, se realizaron 2 pruebas. Para la primera de ellas, se ingreso la cadena (())()(()()(())) de forma manual, cadena para la cual el programa debe indicar que es una cadena válida. La figura 1 muestra el resultado de esta entrada y la figura 2 muestra el contenido del archivo output.txt.

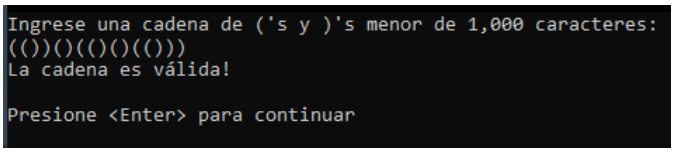

Figura 1. Resultado de ingresar la cadena (())(()()(())) en modo manual. Fuente: elaboración propia.

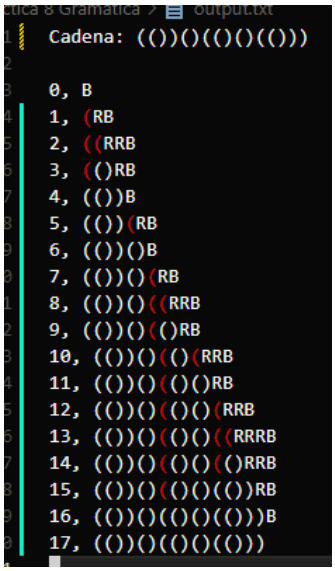

Figura 2. Contenido del archivo output.txt. Fuente: elaboración propia.

Como se observa se obtuvo el resultado esperado. Para la segunda prueba se intento el programa pero en modo aleatorio y como resultado el programa generó una cadena con 286 caracteres y esta no fue válida. El resultado se muestra en la figura 3 y 4 siendo la 4 el contenido del archivo.

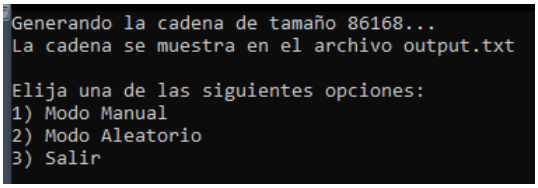

Figura 3. Resultados del programa al generar de forma aleatoria una cadena de 286 caracteres. Fuente: Elaboración propia

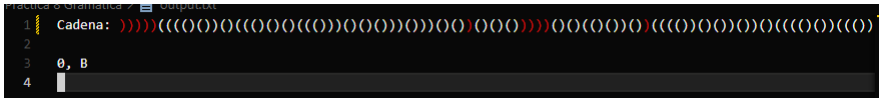

Figura 4. Contenido del archivo output.txt. Fuente: Elaboración propia

5. Conclusión

En esta práctica, se implementó una gramática no ambigua para detectar cadenas de paréntesis sintácticamente correctas. Durante el desarrollo de la práctica, se pudo apreciar la importancia de tener una gramática no ambigua. Esto permitió una interpretación inequívoca de las cadenas de paréntesis, evitando cualquier ambigüedad en su estructura y asegurando que cada cadena pudiera ser generada o derivada de una única manera. La gramática implementada demostró ser efectiva para validar cadenas de paréntesis sintácticamente correctas. Las reglas de producción y las derivaciones permitieron determinar si una cadena cumplía con la estructura correcta de paréntesis, identificando cualquier error sintáctico. Además se destacó la relevancia del análisis sintáctico en el procesamiento de lenguajes formales. El análisis sintáctico permite comprender y validar la estructura de las cadenas de acuerdo con las reglas gramaticales establecidas. En este caso, el análisis sintáctico se centró en garantizar la correcta colocación y equilibrio de los paréntesis.

Referencias

[1] J. E. Hopcroft, *Parsing Techniques: A Practical Guide*. Springer, 2008.