INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Reporte de práctica: Backus-Naur Condicional IF

Grupo:
4CM2

Nombre del alumno:
Maldonado Hernández Alexander

Unidad de aprendizaje:
Teoría de la computación

Maestro:
Genaro Juárez Martínez

Fecha de entrega:
22 de junio de 2025

Índice

1. Introducción

La presente práctica tiene como objetivo implementar un programa capaz de generar derivaciones automáticas a partir de una gramática definida en notación Backus-Naur, centrada en la construcción del condicional `if`. La gramática base utilizada fue la siguiente:

$$\begin{aligned} S &\rightarrow \texttt{iCtSA} \\ A &\rightarrow \texttt{;eS} \mid \varepsilon \end{aligned}$$

El símbolo `S` representa una estructura condicional principal, mientras que `A` modela la posibilidad de una cláusula `else` opcional. El objetivo del programa desarrollado es generar cadenas que respeten esta gramática, controlando la cantidad de condicionales `if` presentes, ya sea de forma manual (a través de una entrada proporcionada por el usuario) o automática (seleccionada aleatoriamente por la máquina).

El programa lleva a cabo derivaciones paso a paso, seleccionando aleatoriamente las producciones a aplicar, hasta alcanzar el número deseado de estructuras `if`. Cada paso de la derivación se registra detalladamente en un archivo de texto, permitiendo al usuario observar el desarrollo completo desde la forma inicial hasta la forma final de la cadena generada. Posteriormente, dicha cadena es procesada para traducir su estructura abstracta a una representación de pseudocódigo, la cual es almacenada en un segundo archivo.

Con el fin de evitar bucles infinitos o un crecimiento descontrolado de las derivaciones, se estableció un límite máximo de 10,000 pasos de derivación. Si este umbral es alcanzado sin cumplir con el número de condicionales deseado, el proceso se detiene con una advertencia al usuario.

La implementación fue desarrollada en el lenguaje de programación Python, haciendo uso de estructuras como listas y funciones recursivas para modelar tanto la expansión de la gramática como la generación del pseudocódigo correspondiente. Esta práctica permitió explorar de manera aplicada el funcionamiento de gramáticas generativas y su utilidad en la construcción de estructuras comunes en lenguajes de programación.

2. Desarrollo

2.1. Diseño del algoritmo de derivación

El diseño del algoritmo de derivación parte del objetivo de construir cadenas válidas según la gramática Backus-Naur proporcionada, garantizando que se generen estructuras sintácticas completas y correctamente anidadas. Para ello, se implementó un mecanismo de expansión aleatoria de reglas, que simula el comportamiento de una derivación no determinista controlada.

La gramática está compuesta por dos reglas principales:

- $S \rightarrow$ `iCtSA`: representa una estructura condicional `if`.
- $A \rightarrow$ `;eS` $\mid \varepsilon$: modela la posibilidad de una cláusula `else`, la cual puede estar presente o no.

El algoritmo inicia con la cadena inicial `S` y, en cada iteración, selecciona aleatoriamente uno de los símbolos no terminales (`S` o `A`) que estén presentes en la cadena actual. Dependiendo de cuál se elija, se aplica una de las siguientes transformaciones:

- Si se selecciona `S`, se sustituye por la expansión `iCtSA`, introduciendo una nueva estructura condicional y un nuevo símbolo `A` al final.
- Si se selecciona `A`, se reemplaza por `;eS` (equivalente a un bloque `else`) o por la cadena vacía (ε), de forma aleatoria.

El proceso se repite hasta que se haya generado la cantidad de condicionales `if` solicitados por el usuario, o hasta que se alcance un límite de derivaciones predefinido (`MAX_DERIVACIONES` = 10,000), establecido para evitar bucles infinitos o la explosión combinatoria de la cadena generada.

Al final del proceso, si aún existen símbolos no terminales `A` en la cadena, estos se eliminan mediante la aplicación forzada de la producción $A \rightarrow \varepsilon$, asegurando así que la cadena final esté completamente compuesta por símbolos terminales.

Este diseño permite un equilibrio entre aleatoriedad y control, generando cadenas variadas pero siempre dentro del marco definido por la gramática, con un número exacto de condicionales `if`, y manteniendo la validez estructural de las expresiones.

2.2. Implementación del generador de derivaciones

La implementación del generador de derivaciones se realizó en el lenguaje de programación Python, debido a su claridad sintáctica y facilidad para manipular cadenas y archivos. El núcleo del programa está compuesto por un conjunto de funciones que operan directamente sobre la cadena en derivación, aplicando reglas gramaticales de forma aleatoria y controlada.

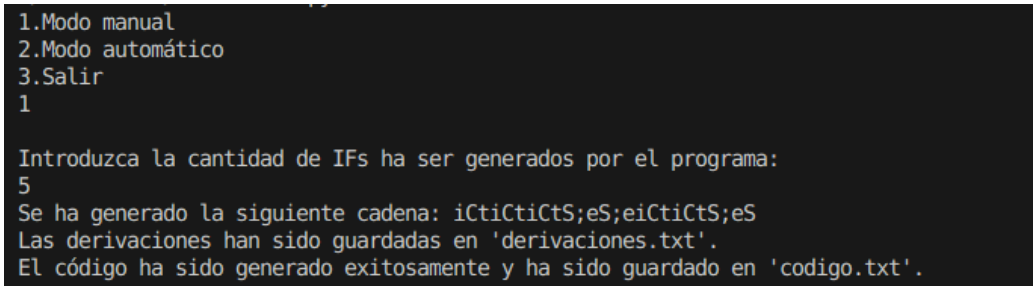

Figura 1: Generando una cadena que contiene 5 IFs

El proceso inicia con la cadena inicial `S`. A partir de ahí, se aplican las siguientes funciones con el objetivo de crear una cadena con la cantidad de IFs solicitados como se puede ver en la Figura 1:

- `derivar_S(cadena)`: reemplaza una ocurrencia del símbolo `S` por la producción `iCtSA`, añadiendo una estructura `if` a la cadena. Esta función solo se invoca mientras no se haya alcanzado la cantidad deseada de condicionales.
- `derivar_A(cadena)`: selecciona aleatoriamente si se desea derivar una ocurrencia del símbolo `A` como `;eS` (generando una cláusula `else`) o como ε (eliminando el símbolo).
- `eliminar_A(cadena)`: se utiliza en la fase final para eliminar cualquier símbolo `A` restante si todavía se está dentro del límite de derivaciones. Esta acción garantiza que la cadena generada no contenga símbolos no terminales.

El motor que coordina estas funciones es `generar_cadena(ifs_deseados)`, el cual realiza un ciclo controlado por dos condiciones: la cantidad de `if`s restantes por generar y el número máximo de derivaciones. Dentro del ciclo, se elige aleatoriamente qué regla aplicar en cada paso, registrando el resultado en un archivo externo llamado `derivaciones.txt`. Este archivo permite llevar un seguimiento detallado y reproducible de todas las transformaciones realizadas sobre la cadena. En la Figura

2 se puede observar las derivaciones que se realizaron para obtener la cadena vista en la Figura 1.

```
1   S
2
3   Regla: S -> iCtSA
4   iCtSA
5
6   Regla: A -> ;eS
7   iCtS;eS
8
9   Regla: S -> iCtSA
10  iCtiCtSA;eS
11
12  Regla: S -> iCtSA
13  iCtiCtiCtSAA;eS
14
15  Regla: A -> ;eS
16  iCtiCtiCtSA;eS;eS
17
18  Regla: S -> iCtSA
19  iCtiCtiCtSA;eiCtSA;eS
20
21  Regla: A -> ;eS
22  iCtiCtiCtS;eS;eiCtSA;eS
23
24  Regla: A -> ε
25  iCtiCtiCtS;eS;eiCtS;eS
26
27  Regla: S -> iCtSA
28  iCtiCtiCtS;eS;eiCtiCtSA;eS
29
30  Regla: A -> ε
31  iCtiCtiCtS;eS;eiCtiCtS;eS
```

Figura 2: Derivaciones para generar una cadena de 5 IFs

En caso de alcanzar el límite de derivaciones antes de cumplir con el objetivo, el programa notifica al usuario y finaliza la ejecución para evitar bloqueos o cadenas incorrectas.

2.3. Conversión a pseudocódigo

Una vez generada la cadena final mediante las derivaciones, el siguiente paso consiste en traducir dicha cadena, que representa la estructura sintáctica del condicional `if`, a una forma legible y comprensible similar al pseudocódigo de un lenguaje de programación. Para ello, se implementaron funciones que analizan recursivamente la cadena, interpretando los símbolos terminales y no terminales según las reglas definidas.

El análisis se centra en detectar patrones específicos, tales como:

- La secuencia `iCt`, que representa el inicio de un condicional `if (condition) { ... }`.
- La presencia del símbolo `;e`, que indica una cláusula `else`.
- Los símbolos `S`, que se traducen en sentencias genéricas `statement;` dentro de los bloques condicionales.

El proceso de conversión se realiza mediante la función `procesar_cadena`, que recorre la cadena y va generando la indentación correspondiente para reflejar la estructura anidada de los bloques condicionales. Para manejar correctamente las subestructuras anidadas, se utiliza la función `extraer_subcadena`, que permite procesar recursivamente las porciones de la cadena delimitadas por paréntesis, garantizando una traducción precisa de las estructuras anidadas.

El resultado de esta conversión se almacena en el archivo `codigo.txt`, donde el usuario puede visualizar una representación estructurada y clara del condicional `if` generado. En la Figura 3 se puede visualizar el pseudocódigo generado para la cadena que se ha estado mostrando durante el reporte.

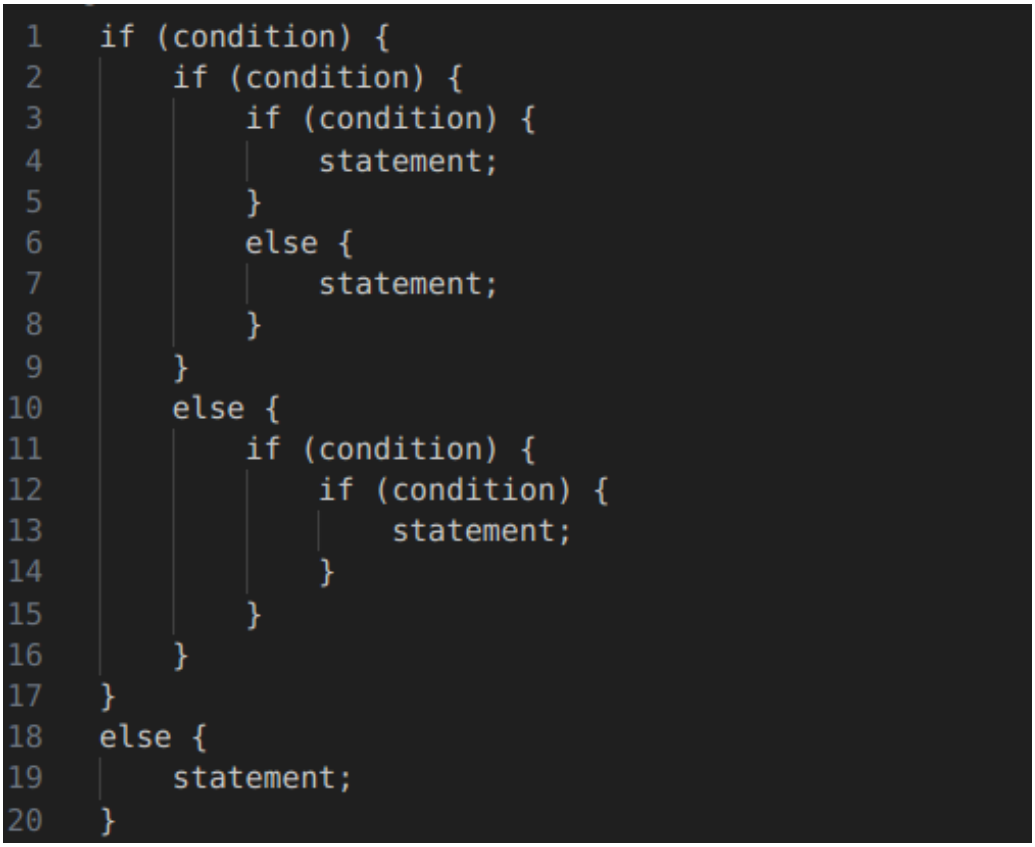

Figura 3: Pseudocódigo de una cadena de 5 IFs

3. Conclusiones

La práctica permitió comprender y aplicar de manera efectiva los conceptos fundamentales relacionados con las gramáticas formales, específicamente en su representación mediante la notación Backus-Naur. A través del desarrollo del programa, se evidenció cómo una estructura gramatical relativamente sencilla puede ser utilizada para generar de forma controlada y aleatoria cadenas que modelan una construcción común en la programación estructurada: el condicional `if-else`.

Uno de los principales aprendizajes fue la importancia de controlar el proceso de derivación para evitar ciclos infinitos o resultados inválidos. La inclusión de un límite máximo de derivaciones y el manejo explícito de los símbolos no terminales aseguraron la correcta finalización del algoritmo y la generación de resultados coherentes. Además, la implementación de modos de ejecución tanto manual como automático facilitó la experimentación con distintas cantidades de estructuras condicionales, favoreciendo el análisis del comportamiento del algoritmo bajo diferentes condiciones.

La conversión de las cadenas generadas a pseudocódigo resultó especialmente útil para visualizar de forma clara la estructura lógica que subyace en las derivaciones, reforzando el vínculo entre teoría formal y práctica de programación. Este paso demostró cómo los lenguajes formales no sólo tienen una utilidad teórica en el análisis de lenguajes, sino también una aplicación directa en la generación automatizada de estructuras válidas en programación.

En resumen, la práctica fortaleció la comprensión de las gramáticas generativas, el diseño de algoritmos de derivación, y su aplicación en contextos reales. Además, fomentó habilidades de abstracción, estructuración lógica y transformación de datos, esenciales para el desarrollo de herramientas automatizadas en ciencias computacionales.

4. Referencias

Daum, B. (2018, noviembre). Backus-Naur Form. https://www.sciencedirect.com/topics/computer-science/backus-naur-form

University, W. M. (2013, agosto). The syntax of C in Backus-Naur Form. https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

5. Anexos

5.1. Anexo 1. Código fuente del programa

```
import random as rd

""" Gramática:
    S -> iCtSA
    A -> ;eS |  """

MAX_DERIVACIONES = 10000

def limpiar_archivos():
    open("derivaciones.txt","w").close()
    open("codigo.txt","w").close()

def derivar_S(cadena):
    posicionesS = [i for i,c in enumerate(cadena) if c == 'S']
    pos = rd.choice(posicionesS)
    nuevaCadena = cadena[:pos] + "(iCtSA)" + cadena[pos+1:] if cadena
    ↪  != 'S' else "iCtSA"
    cadenaImpresa = nuevaCadena.replace('(','').replace(')','')

    with open("derivaciones.txt","a") as d:
        if cadena == 'S':
            d.write("S\n\n")
        d.write("Regla: S -> iCtSA\n")
        d.write(f"{cadenaImpresa}\n\n")

    return nuevaCadena

def derivar_A(cadena):
    posicionesA = [i for i,c in enumerate(cadena) if c == 'A']
    pos = rd.choice(posicionesA)
    decision = rd.choice([';eS',''])
    nuevaCadena = cadena[:pos] + decision + cadena[pos+1:]
    cadenaImpresa =  nuevaCadena.replace('(','').replace(')','')
```

```
    with open("derivaciones.txt","a") as d:
        d.write("Regla: A -> " + ("" if decision == '' else decision)
        ↪  + "\n")
        d.write(f"{cadenaImpresa}\n\n")

    return nuevaCadena

def eliminar_A(cadena):
    posicionesA = [i for i,c in enumerate(cadena) if c == 'A']
    pos = posicionesA[0]
    nuevaCadena = cadena[:pos] + cadena[pos+1:]
    cadenaImpresa =  nuevaCadena.replace('(','').replace(')','')

    with open("derivaciones.txt","a") as d:
        d.write("Regla: A -> \n")
        d.write(f"{cadenaImpresa}\n\n")

    return nuevaCadena

def generar_cadena(ifs_deseados):
    cadena = 'S'
    derivaciones = 0

    #Genera la cantidad de IFs deseada
    while ifs_deseados > 0 and derivaciones < MAX_DERIVACIONES:
        regla = rd.choice(['S','A'])

        if regla == 'A' and 'A' in cadena:
            cadena = derivar_A(cadena)
            derivaciones += 1
        else:
            cadena = derivar_S(cadena)
            ifs_deseados -= 1
            derivaciones += 1

    #La función para si se alcanza la cantidad de derivaciones
    ↪  máxima
    if derivaciones >= MAX_DERIVACIONES and ifs_deseados > 0:
```

```
            print("Se ha alcanzado la cantidad máxima de derivaciones sin
            ↪  lograr generar la cantidad de IFs deseados. Intente con
            ↪  un valor menor.")
            return None

        #Elimina las A restantes de la cadena
        while 'A' in cadena and derivaciones < MAX_DERIVACIONES:
            cadena = eliminar_A(cadena)
            derivaciones += 1

        #La cadena no es valida si se alcanza el máximo de derivaciones
        ↪  y sigue habiendo As en la cadena
        if derivaciones >= MAX_DERIVACIONES and 'A' in cadena:
            print("Se ha alcanzado la cantidad máxima de derivaciones.
            ↪  Intente con un valor menor.")
            return None

        cadenaImpresa =  cadena.replace('(','').replace(')','')
        print(f"Se ha generado la siguiente cadena: {cadenaImpresa}\nLas
        ↪  derivaciones han sido guardadas en 'derivaciones.txt'.")
        return cadena

    def extraer_subcadena(subcadena):
            anidamiento = 1
            i = 1
            contenido = ""

            while i < len(subcadena) and anidamiento > 0:
                if subcadena[i] == '(':
                    anidamiento += 1
                elif subcadena[i] == ')':
                    anidamiento -= 1
                if anidamiento > 0:
                    contenido += subcadena[i]
                i += 1
            return contenido, i  #Contenido interno y caracteres
            ↪  consumidos
```

```
def procesar_cadena(cadena, anidamiento=0):
    i = 0
    codigo = ""

    while i < len(cadena):
        #Encuentra una subcadena "iCt"
        if cadena[i] == 'i' and i+2 < len(cadena) and cadena[i+1] ==
        ↪  'C' and cadena[i+2] == 't':
            codigo += "    " * anidamiento + "if (condition) {\n"
            i += 3

            #Si se tiene un paréntesis, se procesa la subcadena
            ↪  entre paréntesis
            if i < len(cadena) and cadena[i] == '(':
                subcadena, salto = extraer_subcadena(cadena[i:])
                codigo += procesar_cadena(subcadena,anidamiento+1)
                i += salto
            elif i < len(cadena) and cadena[i] == 'S':
                codigo += "    " * (anidamiento+1) + "statement;\n"
                i += 1
            codigo += "    " * anidamiento + "}\n"

        #Encuentra una subcadena ";e"
        elif cadena[i] == ';' and i+1 < len(cadena) and cadena[i+1]
        ↪  == 'e':
            codigo += "    " * anidamiento + "else {\n"
            i += 2

            if i < len(cadena) and cadena[i] == '(':
                subcadena, salto = extraer_subcadena(cadena[i:])
                codigo += procesar_cadena(subcadena,anidamiento+1)
                i += salto
            elif i < len(cadena) and cadena[i] == 'S':
                codigo += "    " * (anidamiento+1) + "statement;\n"
                i += 1
            codigo += "    " * anidamiento + "}\n"

        #Encuentra una subcadena "S"
```

```
        elif cadena[i] == 'S':
            codigo += "    " * anidamiento + "statement;\n"
            i += 1

        else:
            i += 1

    return codigo

def generar_codigo(cadena):
    codigo_procesado = procesar_cadena(cadena)

    with open("codigo.txt","w") as c:
        c.write(codigo_procesado)

    print("El código ha sido generado exitosamente y ha sido guardado
    ↪  en 'codigo.txt'.\n")

def main():
    #Menu inicial
    while True:
        entrada = input("1.Modo manual\n2.Modo
        ↪  automático\n3.Salir\n").strip()

        if entrada == '1':
            while True:
                try:
                    cantidad = int(input("\nIntroduzca la cantidad de
                    ↪  IFs ha ser generados por el programa:\n"))
                    if cantidad > 0:
                        limpiar_archivos()
                        cadena = generar_cadena(cantidad)
                        if cadena == None:
                            continue
                        else:
                            generar_codigo(cadena)
                            break
                    else:
```

```
                        print("Por favor ingrese un número
                        ↪  positivo.")
                        continue
                except ValueError:
                    print("Entrada invalida. Introduzca un número
                    ↪  entero.\n")
                    continue
        elif entrada == '2':
            while True:
                limpiar_archivos()
                cantidad = rd.randint(1,1000)
                print(f"Se generara una cadena con {cantidad} IFs.")
                cadena = generar_cadena(cantidad)
                if cadena == None:
                    continue
                else:
                    generar_codigo(cadena)
                    break

        elif entrada == '3':
            break

        else:
            print("Entrada inválida. Elija una opción: 1, 2 o 3.\n")
            continue

if __name__ == "__main__":
    main()
```