

UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

TEORÍA DE AUTÓMATAS, LENGUAJES Y
COMPUTACIÓN CC321



ESCUELA DE CIENCIAS DE LA COMPUTACIÓN

PRACTICA 4

IMPLEMENTACIÓN DE GRAMÁTICAS Y
GENERADOR PSEUDOALEATORIO

Sección: A

Día y Hora: viernes 21 de noviembre, 2:00 p.m.

Número de práctica: 4

Apellidos y Nombres	Código de alumno
DELGADO ROMERO, Gustavo	20235009B
TORRES REATEGUI, Joaquín	20212661E

Nombre de los Docentes:

- MELCHOR ESPINOZA, Victor Andrés

-

Fecha de entrega del informe: miércoles 26 de noviembre del 2025

2025-II

Introducción

Este informe documenta la implementación de una gramática libre de contexto para generar historias simples y un generador de enteros pseudoaleatorios basado en el algoritmo Park–Miller. Se incluyen descripciones de diseño, fragmentos de código relevantes y ejemplos de salida.

Índice

1. Especificación	3
2. Clase Aleatorio (Park–Miller)	3
3. Clases Regla y Gramatica	3
4. Comportamiento de los contadores	4
5. Ejemplo de salida (ejecutando <code>probar_gramatica()</code>)	5
6. Observaciones y recomendaciones	5

1. Especificación

La gramática utilizada (símbolos no terminales entre <>):

- <inicio>-><historia>
- <historia>-><frase> | <frase>y <historia> | <frase>sino <historia>
- <frase>-><articulo><sustantivo><verbo><articulo><sustantivo>
- <articulo>->el | la | al
- <sustantivo>->gato | niño | perro | niña
- <verbo>->perseguia | besaba

2. Clase Aleatorio (Park–Miller)

La implementación sigue el multiplicador 16807 y módulo $2^{31} - 1 = 2147483647$. Se proporciona un método para generar el siguiente valor y otro para obtener un entero dentro de un límite.

Fragmento de pseudointegers.py

```
class Aleatorio:
    def __init__(self, semilla):
        self.multiplicador = 16807
        self.modulo = 2147483647
        self.actual = semilla
    def siguiente(self):
        self.actual = (self.actual * self.multiplicador) % self.modulo
        return self.actual
    def siguiente_entero(self, limite):
        return self.siguiente() % limite
```

3. Clases Regla y Gramatica

La clase **Regla** almacena **left**, **right** (tupla de símbolos) y **cont** (contador, iniciado en 1). El método **__repr__** devuelve una representación textual del tipo $Q L \rightarrow R1 R2 \dots$.

La clase **Gramatica** mantiene un diccionario de listas de reglas por símbolo izquierdo y un generador pseudoaleatorio. La selección de regla hace lo siguiente:

1. Suma los contadores de las reglas disponibles (**total**).
2. Pide un índice aleatorio en el rango $[0, total - 1]$ mediante **siguiente_entero(total)**.
3. Recorre las reglas restando su **cont** del índice hasta que el índice sea ≤ 0 ; la regla en ese punto es la elegida.
4. Incrementa en 1 el **cont** de todas las reglas NO elegidas (comportamiento intencional documentado en este informe).

Fragmentos de gramatica.py

```

import numpy as np
class Gramatica:
    def __init__(self,seed):
        self.diccionario = {}
        self.aleatorio = Aleatorio(seed)
    def agregar_regla(self,regla):
        if regla.left not in self.diccionario:
            self.diccionario[regla.left] = []
        self.diccionario[regla.left].append(regla)

    def seleccionar(self, left):
        if left not in self.diccionario:
            return tuple()
        reglas = self.diccionario[left]
        total = 0
        for r in reglas:
            total += r.cont

        indice = self.aleatorio.siguiente_entero(total)
        elegido = None

        for regla in reglas:
            indice -= regla.cont

            if indice <=0:
                elegido = regla
                break
        for regla in reglas:
            if regla is not elegido:
                regla.cont+=1

        return elegido.right #type:ignore
    def generar(self, cadena):

        resultado = ""

        if cadena in self.diccionario:
            produccion = self.seleccionar(cadena)
            for simbolo in produccion:

                resultado += self.generar(simbolo)

            else:
                resultado += cadena + " "

        return resultado

    class Regla:
        def __init__(self,izquierda,derecha:tuple):
            self.left = izquierda
            self.right = derecha
            self.cont = 1
        def __repr__(self):
            derecha = " ".join(self.right)

```

4. Comportamiento de los contadores

El diseño incrementa los contadores de las reglas no elegidas, lo que favorece la selección de alternativas diferentes en iteraciones sucesivas: esta es una elección de política deliberada para diversificar las producciones generadas.

5. Ejemplo de salida (ejecutando probar_gramatica())

Debido a la semilla fija (`seed=12345`), la ejecución típica genera tres historias aleatorias y muestra los contadores finales de `<articulo>`. Un ejemplo (ilustrativo) sería:

```
-- Iniciando Prueba de Gramatica --  
  
Generando 3 historias aleatorias:  
  
Historia 1: la perro perseguia el nino  
Historia 2: el nino perseguia al gato sino el nina perseguia al gato y al perro perseguia el perro  
Historia 3: el gato perseguia el perro  
  
Estado de los contadores de <articulo> (deben ser diferentes de 1):  
5 <articulo> -> el  
10 <articulo> -> la  
8 <articulo> -> al
```

6. Observaciones y recomendaciones

- La semilla no está protegida contra el valor 0; si se desea, validar que la semilla esté en $[1, 2^{31} - 2]$.
- La función `generar` añade un espacio tras cada terminal; puede recortarse el resultado final con `strip()` para evitar espacio final.

Conclusión

Se implementó una versión funcional de la gramática y del generador Park–Miller; el diseño de actualización de contadores se documentó como intencional para favorecer la diversidad de producciones. El código incluido es suficiente para replicar las pruebas de la práctica.