

**INSTITUTO POLITECNICO NACIONAL**

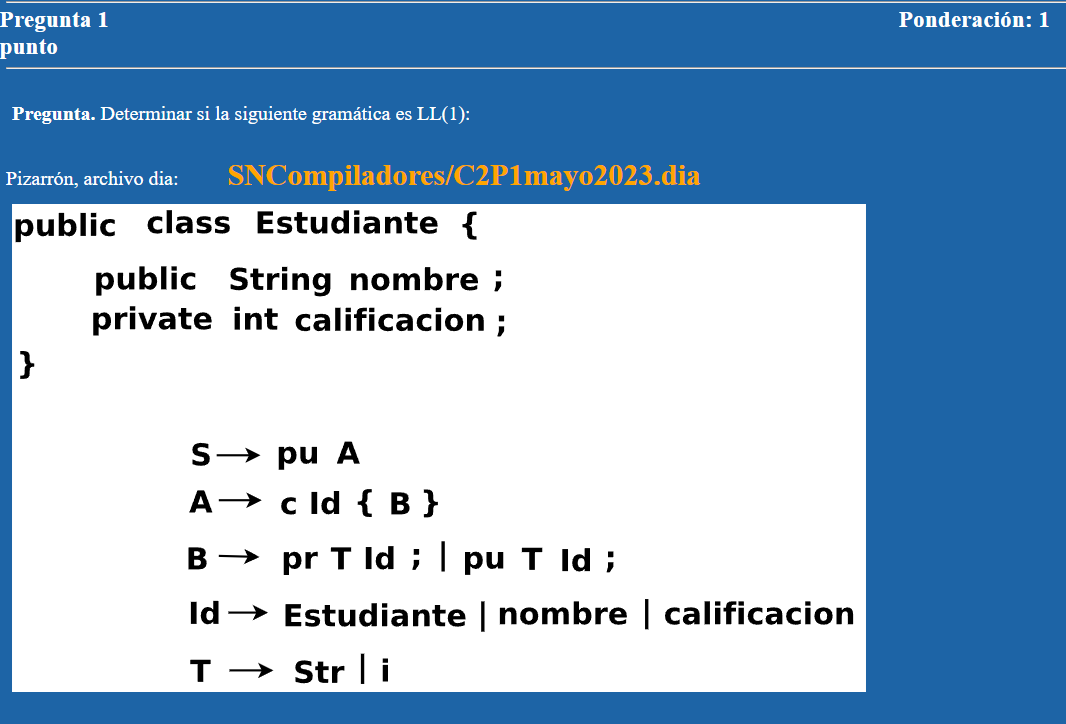
**ESCUELA SUPERIOR DE COMPUTO**

**Compiladores**

**Examen Segundo Departamental**

4CM12

Alvarez Lemus Luis Alejandro



Para determinar si una gramática es LL(1), es necesario examinar las producciones de la gramática y analizar si existe algún conflicto entre los símbolos iniciales de las producciones y los siguientes símbolos.

Primero, vamos a definir los símbolos no terminales:

S: No terminal inicial A: No terminal B: No terminal Id: No terminal T: No terminal

Ahora, vamos a analizar cada producción de la gramática:

S -> pu A No hay conflicto ya que "pu" es único y no tiene ninguna producción adicional.

A -> c Id {B} No hay conflicto ya que "c" es único y no tiene ninguna producción adicional.

B -> pr T Id ; No hay conflicto ya que "pr" es único y no tiene ninguna producción adicional.

B -> pu T Id ; No hay conflicto ya que "pu" es único y no tiene ninguna producción adicional.

Id -> Estudiante No hay conflicto ya que "Estudiante" es único y no tiene ninguna producción adicional.

Id -> nombre No hay conflicto ya que "nombre" es único y no tiene ninguna producción adicional.

Id -> calificacion No hay conflicto ya que "calificacion" es único y no tiene ninguna producción adicional.

T -> Str No hay conflicto ya que "Str" es único y no tiene ninguna producción adicional.

T -> i No hay conflicto ya que "i" es único y no tiene ninguna producción adicional.

En resumen, la gramática proporcionada es LL(1) ya que no hay conflictos entre los símbolos iniciales de las producciones y los siguientes símbolos.

A blue screen with white text

Description automatically generated with low confidence

Para eliminar la recursividad izquierda en la producción B -> B T Id ; | T Id ;, podemos aplicar el siguiente proceso:

Crear un nuevo no terminal B' para reemplazar las producciones recursivas.

Mover las producciones que contienen recursividad a B'.

Agregar una producción en B que derive en B'.

Ajustar las producciones en B' para que no haya recursividad izquierda.

Aplicando estos pasos a la producción B -> B T Id ; | T Id ;, obtendríamos:

B -> T Id B' B' -> ε | ; T Id B'

La producción B' ahora tiene dos opciones: puede derivar en ε (cadena vacía) o puede derivar en ; T Id B', lo que permite la recursión hacia la derecha.

A blue screen with red text

Description automatically generated with low confidence

La gramática libre de contexto (GLC) del código proporcionado se puede obtener de la siguiente manera:

S -> Pu T Id () { return Id ; } Pu -> public T -> String Id -> Nombre

Explicación de las producciones:

S: Representa la producción inicial y corresponde a la estructura del código completo.

Pu: Representa la palabra reservada "public" utilizada en la declaración del método.

T: Representa el tipo de dato de retorno del método, en este caso, "String".

Id: Representa un identificador, en este caso, "Nombre".

Los símbolos "()" y "{}" representan paréntesis y llaves utilizados para delimitar los bloques de código y los argumentos del método.

"return" representa la palabra reservada utilizada para devolver un valor en el método.

Esta gramática libre de contexto describe la estructura del código proporcionado, donde "Pu" representa la visibilidad del método, "T" representa el tipo de dato de retorno, "Id" representa el nombre del método y "()" y "{}" representan la estructura general del método.

A blue screen with red text

Description automatically generated with low confidence

# Código del analizador sintáctico de descenso recursivo

# Función principal del analizador sintáctico

def analizador\_sintactico(tokens):

global indice

indice = 0

setCalificacion(tokens)

# Función para analizar la producción 'setCalificacion'

def setCalificacion(tokens):

if tokens[indice] == 'public':

indice += 1

if tokens[indice] == 'void':

indice += 1

if tokens[indice] == 'setCalificacion':

indice += 1

if tokens[indice] == '(':

indice += 1

if tokens[indice] == 'int':

indice += 1

if tokens[indice] == 'cal':

indice += 1

if tokens[indice] == ')':

indice += 1

if tokens[indice] == '{':

indice += 1

# Llamada a la función asignarValor

asignarValor(tokens)

if tokens[indice] == '}':

indice += 1

# Análisis sintáctico exitoso

print("El análisis sintáctico fue exitoso.")

else:

print("Error de sintaxis. Se esperaba '}'.")

else:

print("Error de sintaxis. Se esperaba '{'.")

else:

print("Error de sintaxis. Se esperaba ')'.")

else:

print("Error de sintaxis. Se esperaba 'cal'.")

else:

print("Error de sintaxis. Se esperaba 'int'.")

else:

print("Error de sintaxis. Se esperaba '('.")

else:

print("Error de sintaxis. Se esperaba 'setCalificacion'.")

else:

print("Error de sintaxis. Se esperaba 'void'.")

else:

print("Error de sintaxis. Se esperaba 'public'.")

# Función para analizar la producción 'asignarValor'

def asignarValor(tokens):

if tokens[indice] == 'calificacion':

indice += 1

if tokens[indice] == '=':

indice += 1

if tokens[indice].isdigit():

indice += 1

if tokens[indice] == ';':

indice += 1

# Asignación de valor exitosa

print("Asignación de valor exitosa.")

else:

print("Error de sintaxis. Se esperaba ';' después del valor asignado.")

else:

print("Error de sintaxis. Se esperaba un número entero.")

else:

print("Error de sintaxis. Se esperaba '='.")

else:

print("Error de sintaxis. Se esperaba 'calificacion'.")

Para utilizar el analizador sintáctico, puedes llamar a la función analizador\_sintactico y pasarle una lista de tokens correspondientes al código que se desea analizar. Cada token debe ser un elemento de la lista. Por ejemplo:

codigo = ['public', 'void', 'setCalificacion', '(', 'int', 'cal', ')', '{', 'calificacion', '=', 'cal', ';', '}']

analizador\_sintactico(codigo)

El analizador sintáctico realizará el análisis de forma recursiva y mostrará mensajes de error en caso de que se encuent