Examen Parcial

Nota

Estudiante	Escuela	Asignatura
Carlos D. Aguilar Chirinos caguilarc@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores Semestre: V

Índice

1. Tarea

1. **Informe:** Elaborar un documento que describa el lenguaje propuesto. El documento debe contener lo siguiente:
   - **Introducción:** Detallen la motivación del lenguaje propuesto y una breve descripción.
   - **Especificación léxica:** Describa cada token y muestre las expresiones regulares.
   - **Gramática:** Muestre la gramática. Para comprobar si la gramática esta bien, puede utilizar esta herramienta (la gramática no debe ser ambigua y debe estar factorizada por la izquierda).
2. **Implementación:** Implementación del analizador sintáctico. La entrada y salida son:
   - **Entrada:** Archivo de texto con código fuente de su lenguaje.
   - **Salida:** Árbol sintáctico (estructura de datos) y archivo de texto con código en Graphviz o Three.js para visualizarlo.
   - El proceso sería:
     a) Desarrolle la gramática del lenguaje (manual).
     b) Implemente un programa para generar la tabla sintáctica a partir de la gramática (código fuente).
     c) Implemente en analizador sintáctico (código fuente):
     - Toma como entrada la tabla sintáctica y código fuente de su lenguaje.
     - Está integrado con el analizador léxico y lo utiliza para generar los *tokens*.
     - Aplica el método LL1.
     - Genera el árbol sintáctico en memoria (como estructura de datos) y además genera un archivo con código en Graphviz o Three.js.

1.1. Entregables

Considere estos dos entregables:

- Se debe elaborar un informe con la descripción del trabajo (*Latex*).
- Archivos de código fuente.

Si entrega otros documentos como: archivos MS Word, archivos comprimidos, etc. tendrá 02 puntos menos.

2. Equipos, materiales y temas utilizados

- Sistema Operativo Windows 11 Pro 23H2 de 64 bits (versión: 22631.2861)
- Visual Studio Code (versión: 1.87.2).
- Procesador AMD Ryzen 5 5600G, RAM 16GB DDR4 2400 MHz.
- Git (versión: 2.44.0).
- Cuenta en GitHub creada con el correo institucional asignado por la Universidad La Salle de Arequipa (caguilarc@ulasalle.edu.pe).
- Conocimientos base en Git.
- Conocimientos base en programación.

3. URL de Repositorio Github

- URL del Repositorio GitHub para clonar o recuperar.
- `https://github.com/CDanielAg/Parcial_Compiladores_24B.git`

4. Actividades con el repositorio GitHub

4.1. Clonar/Actualizar repositorios

Antes de iniciar con la tarea, se tienen que clonar/actualizar los repositorios.

Listing 1: Clonar/Actualizar un repositorio en Git

```
# Para clonar el repositorio:
$ git clone https://github.com/CDanielAg/Parcial_Compiladores_24B.git
# Para actualizar el repositorio:
$ git pull
```

5. Introducción

El desarrollo de software ha sido una de las áreas en las que Python ha demostrado ser un lenguaje de programación sencillo pero efectivo. Sin embargo, su uso de la identificación en el diseño de los bloques de código puede ser un problema para algunos programadores. Para encarar este desafío, hemos creado un nuevo lenguaje de programación basado en Python, pero con la sintaxis propia de lenguajes como C++, y con vocabulario completamente en español. Esto hace que la sintaxis sea más flexible y familiar para personas que están acostumbradas a lenguajes estructurados con llaves, manteniendo algunas de las características poderosas y de fácil acceso de Python.

El lenguaje que estamos desarrollando, aún sin nombre oficial, se caracteriza por las siguientes propiedades clave:

- **Sintaxis en Español:** Todo lo incluido en el lenguaje está pensado para su uso por hispanohablantes, incluyendo las palabras clave y las funciones, que aparecen en español. Esto lo hace más accesible y fácil de aprender para quienes prefieren trabajar en su lengua materna.
- **Encapsulación con Llaves {}:** A semejanza de otros lenguajes de programación, como C++, las funciones, los condicionales y los bucles están definidos utilizando llaves {} en lugar de depender de tabulaciones, como en el caso del lenguaje Python. Esto permite una arquitectura más clara y ordenada en cuanto a la estructura del código.
- **Uso del Símbolo `@` en lugar de `;`:** En este lenguaje, el símbolo `@` reemplaza al símbolo `;` que se usa para separar sentencias en otros lenguajes. Este pequeño pero notable cambio le da al lenguaje una personalidad distintiva.
- **Tokens Definidos:** El lenguaje reconoce una variedad de tokens que permiten escribir código de manera similar a Python, pero con algunas modificaciones y traducciones. Los tokens disponibles incluyen:

```
tokens = (
    'ENTERO', 'FLOTANTE', 'BOOLEANO', 'CADENA',
    'MAS', 'MENOS', 'POR', 'ENTRE', 'IGUAL', 'IGUAL_IGUAL', 'DISTINTO',
    'MENOR', 'MAYOR', 'MENOR_IGUAL', 'MAYOR_IGUAL',
```

```
    'PARENTESIS_ABRIR', 'PARENTESIS_CERRAR', 'CORCHETE_ABRIR', 'CORCHETE_CERRAR',
    'LLAVE_ABRIR', 'LLAVE_CERRAR', 'AT',
    'COMA', 'PUNTO',
    'Y', 'O', 'NO', 'COMENTARIO', 'IDENTIFICADOR', 'IMPRIMIR',
    'SI', 'SINO', 'MIENTRAS', 'PARA', 'DEF', 'RETORNAR',
    'LISTA_ABRIR', 'LISTA_CERRAR', 'DICCIONARIO_ABRIR', 'DICCIONARIO_CERRAR',
    'ROMPER', 'CONTINUAR', 'COMENTAR','SINOSI'
)
```

- **Palabras Reservadas:** Las palabras reservadas en el lenguaje se traducen al español, y son las siguientes:

```
reserved = {
    'if': 'SI',
    'else': 'SINO',
    'while': 'MIENTRAS',
    'for': 'PARA',
    'def': 'DEF',
    'return': 'RETORNAR',
    'break': 'ROMPER',
    'continue': 'CONTINUAR',
    'int': 'ENTERO',
    'float': 'FLOTANTE',
    'boolean': 'BOOLEANO',
    'cadena': 'CADENA',
    'and': 'Y',
    'or': 'O',
    'not': 'NO',
    'True': 'BOOLEANO',
    'False': 'BOOLEANO',
    'print': 'IMPRIMIR',
    'elif' : 'SINOSI'
}
```

Este lenguaje está diseñado para ofrecer una experiencia más accesible y organizada a los desarrolladores hispanohablantes, sin perder la potencia que caracteriza a Python.

6. Especificación Léxica

La especificación léxica de este lenguaje define los tokens que pueden ser reconocidos durante el análisis léxico. A continuación se describen los principales tokens y las expresiones regulares correspondientes:

- **ENTERO:** Representa números enteros, positivos o negativos. La expresión regular es:

  `-?\d+`

- **FLOTANTE:** Representa números de punto flotante, con o sin signo. La expresión regular es:

  `-?\d+\.\d+`

- **BOOLEANO:** Representa valores booleanos (‘True‘ o ‘False‘). La expresión regular es:

  ```
  True|False
  ```

- **CADENA:** Representa secuencias de caracteres entre comillas simples o dobles. La expresión regular es:

  ```
  \'([^\\\']|(\\.))*\'|\"([^\\\"]|(\\.))*\"
  ```

- **OPERADORES ARITMÉTICOS:**
  - **MAS (+):** Suma. La expresión regular es:

    ```
    \+
    ```

  - **MENOS (-):** Resta. La expresión regular es:

    ```
    -
    ```

  - **POR (*):** Multiplicación. La expresión regular es:

    ```
    \*
    ```

  - **ENTRE (/):** División. La expresión regular es:

    ```
    /
    ```

- **OPERADORES RELACIONALES:**
  - **IGUAL (=):** Asignación. La expresión regular es:

    ```
    =
    ```

  - **IGUAL_IGUAL (==):** Comparación de igualdad. La expresión regular es:

    ```
    ==
    ```

  - **DISTINTO (!=):** Comparación de desigualdad. La expresión regular es:

    ```
    !=
    ```

  - **MAYOR (¿):** Mayor que. La expresión regular es:

    ```
    >
    ```

  - **MENOR (¡):** Menor que. La expresión regular es:

    ```
    <
    ```

  - **MENOR_IGUAL (¡=):** Menor o igual que. La expresión regular es:

```
<=
```

- **MAYOR_IGUAL (¿=):** Mayor o igual que. La expresión regular es:

```
>=
```

- **DELIMITADORES:**
  - **PARENTESIS_ABRIR (():** La expresión regular es:

```
\(
```

  - **PARENTESIS_CERRAR ()):** La expresión regular es:

```
\)
```

  - **LLAVE_ABRIR {}:** La expresión regular es:

```
\{
```

  - **LLAVE_CERRAR {}:** La expresión regular es:

```
\}
```

  - **AT (@):** Separador de sentencias. La expresión regular es:

```
@
```

- **IDENTIFICADOR:** Representa nombres de variables y funciones. La expresión regular es:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

- **COMENTARIO:** Comentarios de una sola línea. La expresión regular es:

```
\#.*
```

Cada uno de estos tokens es clave para el análisis léxico del lenguaje, permitiendo que el compilador identifique correctamente las estructuras del código y lo traduzca en acciones concretas para su ejecución.

7. Gramatíca

En la implementación de compiladores, el paso del análisis sintáctico es uno de los más críticos y propósito del mismo es comprobar la correcta estructura del código fuente para con el lenguaje de programación seleccionado. La gramática que se presenta a continuación, describe las reglas de estructura de un lenguaje simple y bien organizar , que podría ser enseñado con el fin de guardar propósitos fundamentales como la recursión, la jerarquía de los operadores o las estructuras de control.

Este lenguaje está diseñado para apoyar la enseñanza de los siguientes conceptos fundamentales:

- **Instrucciones básicas:** Operaciones comunes como la asignación, impresión y las estructuras de control como los bucles `mientras` y las condicionales `si-sino`.
- **Funciones y parámetros:** Definición de funciones con parámetros, brindando una comprensión clara de cómo se estructura el flujo de un programa.
- **Expresiones aritméticas y lógicas:** Uso de operadores básicos (`+`, `-`, `*`, `/`) y operadores de comparación (`==`, `!=`, `<`, `>`) que permiten realizar cálculos y decisiones en el programa.
- **Control de flujo:** Bucle `mientras` y estructura condicional que permite crear programas con comportamiento dinámico, dependiendo de las condiciones.
- **Jerarquía en las expresiones:** La gramática incluye reglas para operadores aritméticos y lógicos, introduciendo la jerarquía de operaciones para evaluar expresiones complejas.

```
PROGRAMA -> INSTRUCCIONES

INSTRUCCIONES -> INSTRUCCION INSTRUCCIONES
INSTRUCCIONES -> ''

INSTRUCCION -> ASIGNACION AT
INSTRUCCION -> Imprimir AT
INSTRUCCION -> Mientras
INSTRUCCION -> FUNCION
INSTRUCCION -> RETURN EXPRESION
INSTRUCCION -> CONDICIONAL
INSTRUCCION -> ROMPER AT

CONDICIONAL -> SI PARENTESIS_ABRIR CONDICION PARENTESIS_CERRAR LLAVE_ABRIR INSTRUCCIONES
    LLAVE_CERRAR CONDICIONAL'
CONDICIONAL' -> SINO LLAVE_ABRIR INSTRUCCIONES LLAVE_CERRAR
CONDICIONAL' -> SINOSI PARENTESIS_ABRIR CONDICION PARENTESIS_CERRAR LLAVE_ABRIR
    INSTRUCCIONES LLAVE_CERRAR CONDICIONAL'
CONDICIONAL' -> ''

FUNCION -> DEF IDENTIFICADOR PARENTESIS_ABRIR PARAMETROS PARENTESIS_CERRAR LLAVE_ABRIR
    INSTRUCCIONES LLAVE_CERRAR

PARAMETROS -> ''
PARAMETROS -> IDENTIFICADOR PARAMETROS'

PARAMETROS' -> COMA IDENTIFICADOR PARAMETROS'
PARAMETROS' -> ''

Imprimir -> IMPRIMIR PARENTESIS_ABRIR IMPRIMIR' PARENTESIS_CERRAR

IMPRIMIR' -> ''
IMPRIMIR' -> EXPRESION MASEXPRESION

Mientras -> MIENTRAS PARENTESIS_ABRIR CONDICION PARENTESIS_CERRAR LLAVE_ABRIR
    INSTRUCCIONES LLAVE_CERRAR

CONDICION -> EXPRESION CONDICION'

CONDICION' -> OPERADORLOG EXPRESION CONDICION'
CONDICION' -> ''
```

```
MASEXPRESION -> COMA EXPRESION MASEXPRESION
MASEXPRESION -> ''

ASIGNACION -> IDENTIFICADOR IGUAL EXPRESION

EXPRESION -> FACTOR EXPRESION'
EXPRESION' -> OPERADOR FACTOR
EXPRESION' -> COMPARACION FACTOR
EXPRESION' -> ''

OPERADOR -> MAS
OPERADOR -> MENOS
OPERADOR -> ENTRE
OPERADOR -> POR

OPERADORLOG -> Y
OPERADORLOG -> O
OPERADORLOG -> NO

FACTOR -> IDENTIFICADOR
FACTOR -> ENTERO
FACTOR -> FLOTANTE
FACTOR -> BOOLEANO
FACTOR -> CADENA

COMPARACION -> IGUAL_IGUAL
COMPARACION -> DISTINTO
COMPARACION -> MENOR
COMPARACION -> MENOR_IGUAL
COMPARACION -> MAYOR
COMPARACION -> MAYOR_IGUAL
```

8. Tabla sintáctica

A continuación, se describe brevemente cada una de las funciones principales incluidas en el código:

- `read_grammar`: Lee las reglas de una gramática desde un archivo y las almacena en una lista.
- `collect_alphabet_and_nonterminals`: Recopila el alfabeto completo, los no terminales y terminales presentes en la gramática.
- `collect_firsts`: Calcula el conjunto `FIRST` para cada no terminal.
- `collect_follows`: Calcula el conjunto `FOLLOW` para cada no terminal.
- `make_rule_table`: Construye la tabla de análisis sintáctico LL(1) usando los conjuntos `FIRST` y `FOLLOW`.
- `write_csv`: Escribe la tabla generada en un archivo CSV.
- `write_nonterminals`: Guarda los no terminales en un archivo de texto.

Listing 2: Tabla*sintactica.py*

```
import csv
import os

# Definir el simbolo EPSILON, que representa una produccion vacia
epsilon = "''"

# Leer la gramatica desde un archivo y almacenar las reglas en una lista
def read_grammar(file_path):
    rules = []
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            if line:
                rules.append(line)
    return rules

# Recopilar el alfabeto, los no terminales y los terminales de la gramatica
def collect_alphabet_and_nonterminals(rules):
    alphabet = set()
    nonterminals = set()
    for rule in rules:
        left, right = rule.split('->')
        nonterminal = left.strip()
        nonterminals.add(nonterminal)
        symbols = right.strip().split()
        alphabet.update(symbols)
    terminals = alphabet - nonterminals
    return list(alphabet), list(nonterminals), list(terminals)

# Calcular los conjuntos FIRST para cada no terminal
def collect_firsts(rules, nonterminals, terminals):
    firsts = {nt: set() for nt in nonterminals}
    not_done = True
    while not_done:
        not_done = False
        for rule in rules:
            left, right = rule.split('->')
            nonterminal = left.strip()
            symbols = right.strip().split()
            if symbols[0] == epsilon:
                not_done |= epsilon not in firsts[nonterminal]
                firsts[nonterminal].add(epsilon)
            else:
                for symbol in symbols:
                    if symbol in terminals or symbol == epsilon:
                        not_done |= symbol not in firsts[nonterminal]
                        firsts[nonterminal].add(symbol)
                        break
                    else:
                        old_size = len(firsts[nonterminal])
                        firsts[nonterminal].update(firsts[symbol] - {epsilon})
                        not_done |= len(firsts[nonterminal]) > old_size
                        if epsilon not in firsts[symbol]:
                            break
    return firsts

# Calcular los conjuntos FOLLOW para cada no terminal
def collect_follows(rules, nonterminals, firsts):
    follows = {nt: set() for nt in nonterminals}
    # Agregar el simbolo de fin de cadena ('$') al conjunto FOLLOW del simbolo inicial
    follows[rules[0].split('->')[0].strip()].add('$')
    not_done = True
    while not_done:
        not_done = False
        for rule in rules:
```

```
                left, right = rule.split('->')
                nonterminal = left.strip()
                symbols = right.strip().split()
                for i, symbol in enumerate(symbols):
                    if symbol in nonterminals:
                        follows_set = follows[symbol]
                        if i + 1 < len(symbols):
                            next_symbol = symbols[i + 1]
                            if next_symbol in nonterminals:
                                follows_set.update(firsts[next_symbol] - {epsilon})
                            else:
                                follows_set.add(next_symbol)
                        # Si es el ultimo simbolo o tiene epsilon en su FIRST, agregar FOLLOW del no terminal
                        if i + 1 == len(symbols) or epsilon in firsts.get(next_symbol, []):
                            old_size = len(follows_set)
                            follows_set.update(follows[nonterminal])
                            not_done |= len(follows_set) > old_size
        return follows

    # Generar la tabla de analisis sintactico LL(1)
    def make_rule_table(rules, nonterminals, terminals, firsts, follows):
        rule_table = {nt: {t: '' for t in terminals + ['$']} for nt in nonterminals}
        for rule in rules:
            left, right = rule.split('->')
            nonterminal = left.strip()
            symbols = right.strip().split()
            development_firsts = collect_firsts_for_development(symbols, firsts, terminals)
            for symbol in development_firsts:
                if symbol != epsilon:
                    rule_table[nonterminal][symbol] = f"{nonterminal} -> {right.strip()}"
            # Agregar la produccion a los FOLLOW si epsilon esta en FIRST
            if epsilon in development_firsts:
                for follow_symbol in follows[nonterminal]:
                    rule_table[nonterminal][follow_symbol] = f"{nonterminal} -> {right.strip()}"
        return rule_table

    # Calcular el conjunto FIRST para una secuencia de simbolos (produccion)
    def collect_firsts_for_development(development, firsts, terminals):
        result = set()
        for symbol in development:
            if symbol in terminals:
                result.add(symbol)
                break
            result.update(firsts[symbol] - {epsilon})
            if epsilon not in firsts[symbol]:
                break
        else:
            result.add(epsilon)
        return result

    # Escribir la tabla de analisis en un archivo CSV
    def write_csv(rule_table, output_file):
        with open(output_file, 'w', newline='') as csvfile:
            writer = csv.writer(csvfile)
            header = ['Nonterminal'] + list(rule_table[next(iter(rule_table))].keys())
            writer.writerow(header)
            for nonterminal, rules in rule_table.items():
                row = [nonterminal] + [rules[terminal] for terminal in header[1:]]
                writer.writerow(row)

    # Escribir los no terminales en un archivo de texto
    def write_nonterminals(nonterminals, output_file):
        with open(output_file, 'w') as file:
            for nonterminal in nonterminals:
```

```
                file.write(nonterminal + '\n')

    # Funcion principal para ejecutar el programa
    def main():
        grammar_file = 'Gramatica.txt' # Nombre del archivo con la gramtica
        output_file = 'll1_table.csv' # Nombre del archivo de salida CSV
        nonterminals_file = 'no_terminales.txt' # Nombre del archivo de salida para no terminales

        # Verificar si el archivo de gramatica existe
        if not os.path.exists(grammar_file):
            print(f"Error: El archivo {grammar_file} no existe.")
            return

        # Leer la gramatica y calcular los conjuntos FIRST y FOLLOW
        rules = read_grammar(grammar_file)
        alphabet, nonterminals, terminals = collect_alphabet_and_nonterminals(rules)
        firsts = collect_firsts(rules, nonterminals, terminals)
        follows = collect_follows(rules, nonterminals, firsts)
        # Generar la tabla LL(1) y escribirla en un archivo CSV
        rule_table = make_rule_table(rules, nonterminals, terminals, firsts, follows)
        write_csv(rule_table, output_file)
        # Escribir los no terminales en un archivo de texto
        write_nonterminals(nonterminals, nonterminals_file)
        print(f"Tabla LL(1) generada y guardada en {output_file}")
        print(f"No terminales guardados en {nonterminals_file}")

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

9. Analizador sintáctico

La técnica llamada análisis predictivo LL(1) de la sintaxis se considera un instrumento fundamental mientras se construye el compilador o el analizador de algún lenguaje formal. Este tipo de analizador emplea una tabla de análisis para conducir el proceso de derivación desde una cadena de tokens de entrada, el cual se produce en la etapa de análisis léxico. El código presentado aquí contiene un parser LL(1) en Python que carga una gramática formal y una lista de tokens desde archivos. También utiliza una tabla LL(1) pre-construida para realizar el análisis de manera sistemática y eficiente.

Este parser realiza las siguientes tareas:

- Carga la gramática y los tokens de archivos especificados.
- Calcula el conjunto de símbolos de la gramática (alfabeto, no terminales y terminales).
- Carga la tabla de análisis sintáctico LL(1) desde un archivo CSV.
- Procesa la entrada, aplicando reglas de la tabla LL(1) paso a paso, verificando la correcta derivación de los tokens.
- Exporta el rastreo del análisis a un archivo CSV para un análisis detallado del proceso de parsing.

Listing 3: AnalisadorSintactico.py

```
import csv
import re

EPSILON = "''"

class LLParser:
    def __init__(self, grammar_file, tokens_file):
```

```
8               self.alphabet = []
9               self.nonterminals = []
10              self.terminals = []
11              self.rules = []
12              self.tokens = []
13              self.rule_table = {}
14
15              self._load_grammar(grammar_file)
16              self._load_tokens(tokens_file)
17              self._collect_alphabet_and_symbols()
18              self._load_rule_table()
19
20          def _load_grammar(self, grammar_file):
21              with open(grammar_file, 'r') as f:
22                  self.rules = [line.strip() for line in f if '->' in line]
23
24          def _load_tokens(self, tokens_file):
25              with open(tokens_file, 'r') as f:
26                  self.tokens = f.read().strip().split()
27
28          def _collect_alphabet_and_symbols(self):
29              for rule in self.rules:
30                  lhs, rhs = rule.split('->')
31                  nonterminal = lhs.strip()
32                  development = rhs.strip().split()
33
34                  if nonterminal not in self.nonterminals:
35                      self.nonterminals.append(nonterminal)
36
37                  for symbol in development:
38                      if symbol != EPSILON:
39                          if symbol not in self.alphabet:
40                              self.alphabet.append(symbol)
41
42              self.terminals = [symbol for symbol in self.alphabet if symbol not in self.nonterminals]
43
44          def _load_rule_table(self):
45              # Load the rule table from an external CSV file
46              with open('ll1_table.csv', mode='r') as file:
47                  csv_reader = csv.DictReader(file)
48                  for row in csv_reader:
49                      nonterminal = row['Nonterminal']
50                      self.rule_table[nonterminal] = {}
51                      for terminal, rule in row.items():
52                          if terminal != 'Nonterminal' and rule:
53                              self.rule_table[nonterminal][terminal] = rule
54
55          def parse_input(self):
56              stack = ['$', self.nonterminals[0]]
57              index = 0
58              input_tokens = self.tokens + ['$']
59
60              rows = []
61              while len(stack) > 0:
62                  top = stack.pop()
63                  current_token = input_tokens[index]
64                  rule = self.rule_table.get(top, {}).get(current_token) if top in self.nonterminals
                        else ""
65
66                  rows.append([" ".join(stack), " ".join(input_tokens[index:]), f"{top} ->
                        {rule.split('->')[1].strip()}" if rule else "Accept" if top == '$' and
                        current_token == '$' else ""])
67
68                  if top == current_token:
69                      index += 1
```

```
                elif top in self.terminals or top == '$':
                    rows.append(["Error: terminal mismatch."])
                    break
                elif top in self.nonterminals:
                    if rule is None:
                        rows.append([f"Error: no rule for nonterminal '{top}' with token
                            '{current_token}'"])
                        break
                    _, rhs = rule.split('->')
                    symbols = rhs.strip().split()
                    if symbols != [EPSILON]:
                        stack.extend(reversed(symbols))
                else:
                    rows.append(["Error: unknown symbol on stack."])
                    break

            self._export_parsing_process_to_csv("rastreo.csv", rows)

        def _export_parsing_process_to_csv(self, csv_filename, rows):
            with open(csv_filename, 'w', newline='') as csvfile:
                csv_writer = csv.writer(csvfile)
                header = ["Stack", "Input", "Rule"]
                csv_writer.writerow(header)
                for row in rows:
                    csv_writer.writerow(row)

    if __name__ == "__main__":
        parser = LLParser("Gramatica.txt", "tokens.txt")
        parser.parse_input()
```

10. Arbol sintáctico

Este código está diseñado para procesar el rastreo del análisis de una entrada mediante un parser LL(1). A partir de este rastreo, construye un árbol que representa la aplicación de las reglas de la gramática en cada paso del análisis sintáctico. El árbol se genera de manera visual utilizando la herramienta `graphviz`, y se exporta como una imagen en formato PNG.

El código realiza las siguientes funciones principales:

- **Cargar el rastreo**: Lee las reglas aplicadas durante el proceso de análisis desde un archivo CSV.
- **Generar el árbol sintáctico**: Construye un árbol a partir del rastreo, creando nodos para cada no terminal y terminal según las reglas de la gramática.
- **Agregar nodos epsilon**: Añade nodos epsilon (ϵ) para representar producciones vacías en nodos hoja que corresponden a no terminales.
- **Resaltar hojas sin hijos**: Resalta los nodos hoja del árbol, que no tienen hijos, con un color de relleno amarillo para facilitar la visualización.
- **Exportar el árbol**: Genera un archivo de imagen PNG que contiene la representación visual del árbol sintáctico.

Listing 4: Generar$_a$*rbol.py*

```
    import csv
    from graphviz import Digraph

    class Nodo:
        def __init__(self, etiqueta, identificador):
```

```
            self.etiqueta = etiqueta
            self.identificador = identificador
            self.hijos = []

        def agregar_hijo(self, hijo):
            self.hijos.append(hijo)

    def cargar_rastreo(nombre_archivo):
        rastreo = []
        with open(nombre_archivo, mode='r') as archivo:
            lector = csv.DictReader(archivo)
            for fila in lector:
                rastreo.append(fila)
        return rastreo

    def generar_arbol_sintactico(rastreo, nombre_archivo):
        dot = Digraph(comment='rbol Sintctico')
        contador_nodos = 0
        nodos = {}
        reglas_diccionario = {}

        # Crear un nodo raz para comenzar el rbol
        raiz = None

        # Crear nodos usando las reglas de la traza
        for entrada in rastreo:
            regla = entrada['Rule']
            if '->' in regla:
                cabeza, produccion = regla.split('->')
                cabeza = cabeza.strip()
                simbolos_produccion = [simbolo for simbolo in produccion.strip().split() if simbolo
                    != "''"]

                # Almacenar la regla en el diccionario sin sobrescribir reglas existentes
                if cabeza in reglas_diccionario:
                    if simbolos_produccion not in reglas_diccionario[cabeza]:
                        reglas_diccionario[cabeza].append(simbolos_produccion)
                else:
                    reglas_diccionario[cabeza] = [simbolos_produccion]

                # Crear un nodo para la cabeza si no existe
                if cabeza not in nodos:
                    nodo_cabeza = Nodo(cabeza, f"N{contador_nodos}")
                    nodos[cabeza] = nodo_cabeza
                    dot.node(nodo_cabeza.identificador, cabeza)
                    contador_nodos += 1
                    if raiz is None:
                        raiz = nodo_cabeza

                # Obtener el nodo cabeza actual
                nodo_cabeza = nodos[cabeza]

                # Crear nodos para cada smbolo en la produccin y conectarlos correctamente
                for simbolo in simbolos_produccion:
                    identificador_nodo_simbolo = f"{simbolo}_{contador_nodos}"
                    nodo_simbolo = Nodo(simbolo, identificador_nodo_simbolo)
                    nodos[identificador_nodo_simbolo] = nodo_simbolo
                    dot.node(nodo_simbolo.identificador, simbolo)
                    contador_nodos += 1

                    # Conectar el nodo cabeza con el nodo smbolo
                    nodo_cabeza.agregar_hijo(nodo_simbolo)
                    dot.edge(nodo_cabeza.identificador, nodo_simbolo.identificador)

                    # Asegurarse de que los nodos hijos tambin puedan tener relaciones correctas
```

```
                    nodos[simbolo] = nodo_simbolo

        # Verificar y agregar nodo epsilon para nodos hoja sin hijos que estn en no_terminales.txt
        agregar_epsilon_a_hojas_sin_hijos(nodos, 'no_terminales.txt', dot)

        # Resaltar nodos hojas sin hijos con relleno amarillo
        resaltar_hojas_sin_hijos(nodos, dot)

        dot.render(nombre_archivo, format='png', cleanup=True)
        print(f"rbol  sintctico  guardado en {nombre_archivo}.png")

    def agregar_epsilon_a_hojas_sin_hijos(nodos, no_terminales_file, dot):
        with open(no_terminales_file, 'r') as file:
            no_terminales = {line.strip() for line in file}

        for nodo in nodos.values():
            if not nodo.hijos and nodo.etiqueta in no_terminales:
                # Crear un nodo hijo con el smbolo epsilon
                identificador_nodo_epsilon = f"epsilon_{nodo.identificador}"
                nodo_epsilon = Nodo("ε", identificador_nodo_epsilon)
                nodo.agregar_hijo(nodo_epsilon)
                dot.node(nodo_epsilon.identificador, "ε", style='filled', fillcolor='yellow')
                dot.edge(nodo.identificador, nodo_epsilon.identificador)

    def resaltar_hojas_sin_hijos(nodos, dot):
        for nodo in nodos.values():
            if not nodo.hijos:
                # Resaltar el nodo hoja con relleno amarillo
                dot.node(nodo.identificador, nodo.etiqueta, style='filled', fillcolor='yellow')

    def main():
        nombre_archivo_rastreo = 'rastreo.csv'
        nombre_archivo_arbol = 'arbol_sintactico'

        rastreo = cargar_rastreo(nombre_archivo_rastreo)
        generar_arbol_sintactico(rastreo, nombre_archivo_arbol)

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

11. Analizador léxico

Este lexer identifica varios tipos de tokens, incluyendo operadores aritméticos, comparativos y lógicos, así como palabras reservadas del lenguaje como `si`, `sino`, `mientras`, `para`, y otros elementos comunes de lenguajes de programación como identificadores, números enteros, flotantes, cadenas de texto, y comentarios.

El lexer realiza las siguientes funciones principales:

- **Definición de tokens**: Identifica tokens básicos como operadores matemáticos (`+`, `-`, `*`, `/`), operadores comparativos (`==`, `!=`, `<`, `>`) y estructuras de control (`si`, `sino`, `mientras`, etc.).
- **Manejo de tipos de datos**: Reconoce tipos de datos como enteros, flotantes, booleanos y cadenas.
- **Ignorar espacios y tabulaciones**: El lexer está diseñado para ignorar los espacios en blanco y las tabulaciones para centrarse solo en los tokens significativos.
- **Detección de errores léxicos**: Si encuentra caracteres no válidos, los ignora y emite un mensaje de advertencia.

- **Prueba de lexing**: El código incluye un ejemplo para probar la funcionalidad del lexer y guardar los tokens generados en un archivo.

Listing 5: LexerPythonES.py

```
import ply.lex as lex

# Lista de tokens
tokens = (
    'ENTERO', 'FLOTANTE', 'BOOLEANO', 'CADENA',
    'MAS', 'MENOS', 'POR', 'ENTRE', 'IGUAL', 'IGUAL_IGUAL', 'DISTINTO',
    'MENOR', 'MAYOR', 'MENOR_IGUAL', 'MAYOR_IGUAL',
    'PARENTESIS_ABRIR', 'PARENTESIS_CERRAR', 'CORCHETE_ABRIR', 'CORCHETE_CERRAR',
    'LLAVE_ABRIR', 'LLAVE_CERRAR', 'AT',
    'COMA', 'PUNTO',
    'Y', 'O', 'NO', 'COMENTARIO', 'IDENTIFICADOR', 'IMPRIMIR',
    'SI', 'SINO', 'MIENTRAS', 'PARA', 'DEF', 'RETORNAR',
    'LISTA_ABRIR', 'LISTA_CERRAR', 'DICCIONARIO_ABRIR', 'DICCIONARIO_CERRAR',
    'ROMPER', 'CONTINUAR', 'COMENTAR','SINOSI'
)

# Palabras reservadas
reserved = {
    'if': 'SI',
    'else': 'SINO',
    'while': 'MIENTRAS',
    'for': 'PARA',
    'def': 'DEF',
    'return': 'RETORNAR',
    'break': 'ROMPER',
    'continue': 'CONTINUAR',
    'int': 'ENTERO',
    'float': 'FLOTANTE',
    'boolean': 'BOOLEANO',
    'cadena': 'CADENA',
    'and': 'Y',
    'or': 'O',
    'not': 'NO',
    'True': 'BOOLEANO',
    'False': 'BOOLEANO',
    'print': 'IMPRIMIR',
    'elif' : 'SINOSI'
}

# Reglas de expresiones regulares para tokens simples
t_AT = r'@'
t_MAS = r'\+'
t_MENOS = r'-'
t_POR = r'\*'
t_ENTRE = r'/'
t_IGUAL = r'='
t_IGUAL_IGUAL = r'=='
t_DISTINTO = r'!='
t_MENOR = r'<'
t_MAYOR = r'>'
t_MENOR_IGUAL = r'<='
t_MAYOR_IGUAL = r'>='
t_PARENTESIS_ABRIR = r'\('
t_PARENTESIS_CERRAR = r'\)'
t_CORCHETE_ABRIR = r'\['
t_CORCHETE_CERRAR = r'\]'
t_LLAVE_ABRIR = r'\{'
t_LLAVE_CERRAR = r'\}'
t_LISTA_ABRIR = r'\['
```

```
60      t_LISTA_CERRAR = r'\]'
61      t_DICCIONARIO_ABRIR = r'\{'
62      t_DICCIONARIO_CERRAR = r'\}'
63      t_COMA = r','
64      t_PUNTO = r'\.'
65
66      # Operadores lgicos
67      t_Y = r'y'
68      t_O = r'o'
69      t_NO = r'no'
70
71      # Definicin de las reglas de los tokens ms complejos
72      def t_FLOTANTE(t):
73          r'-?\d+\.\d+'
74          t.value = float(t.value)
75          return t
76
77      def t_ENTERO(t):
78          r'-?\d+'
79          t.value = int(t.value)
80          return t
81
82      def t_BOOLEANO(t):
83          r'True|False'
84          t.value = True if t.value == 'True' else False
85          return t
86
87      def t_CADENA(t):
88          r'\'([^\\\']|(\\.))*\'|\"([^\\\"]|(\\.))*\"'
89          t.value = t.value[1:-1]
90          return t
91
92      def t_IDENTIFICADOR(t):
93          r'[a-zA-Z_][a-zA-Z_0-9]*'
94          t.type = reserved.get(t.value, 'IDENTIFICADOR')
95          return t
96
97      def t_COMENTARIO(t):
98          r'\#.*'
99          pass
100
101     def t_newline(t):
102         r'\n'
103         t.lexer.lineno += 1
104
105     t_ignore = ' \t'
106
107     def t_error(t):
108         print(f"Carcter ilegal: {t.value[0]}")
109         t.lexer.skip(1)
110
111     # Construir el lexer
112     lexer = lex.lex()
113
114     # Prueba del lexer con condicional
115     data = '''
116     def f(x,g(c,7),h(g,k,l())) {
117     }
118     '''
119
120     lexer.input(data)
121
122     # Tokenizar e imprimir solo los tipos de tokens separados por un espacio
123     tokens_list = []
124
```

```
        while True:
            tok = lexer.token()
            if not tok:
                break
            tokens_list.append(tok.type)

        # Guardar los tokens en un archivo tokens.txt
        with open("tokens.txt", "w") as file:
            file.write(" ".join(tokens_list))
```

12. Ejemplos de código:

En esta sección se dan ejemplos de código que incluyen todos los componentes principales para desarrollar un compilador o intérprete a partir del analizador léxico y la generación de árboles de análisis. Los ejemplos son de algunos módulos que son un lexer, analizador sintáctico del tipo LL(1) y un constructor de árbol de sintaxis. Igualmente, estos fragmentos de código tienen como una intención mostrar cómo se aplican desta forma y cómo se utilizan estos instrumentos fundamentales en el procesamiento de lenguajes formales.

Listing 6: Ejmplo 01

```
        #Asignacion de variable
        hola = 1 @
```

Stack	**Input**	**Rule**
$	IDENTIFICADOR IGUAL ENTERO AT $	PROGRAMA -¿INSTRUCCIONES
$	IDENTIFICADOR IGUAL ENTERO AT $	INSTRUCCIONES -¿INSTRUCCION INSTRUCCIONES
$ INSTRUCCIONES	IDENTIFICADOR IGUAL ENTERO AT $	INSTRUCCION -¿ASIGNACION AT
$ INSTRUCCIONES AT	IDENTIFICADOR IGUAL ENTERO AT $	ASIGNACION -¿IDENTIFICADOR IGUAL EXPRESION
$ INSTRUCCIONES AT EXPRESION IGUAL	IDENTIFICADOR IGUAL ENTERO AT $	
$ INSTRUCCIONES AT EXPRESION	IGUAL ENTERO AT $	
$ INSTRUCCIONES AT	ENTERO AT $	EXPRESION -¿FACTOR EXPRESION’
$ INSTRUCCIONES AT EXPRESION’	ENTERO AT $	FACTOR -¿ENTERO
$ INSTRUCCIONES AT EXPRESION’	ENTERO AT $	
$ INSTRUCCIONES AT	AT $	EXPRESION’ -¿”
$ INSTRUCCIONES	AT $	
$	$	INSTRUCCIONES -¿”
, $	Accept	

Tabla 1: Proceso de análisis sintáctico LL(1)

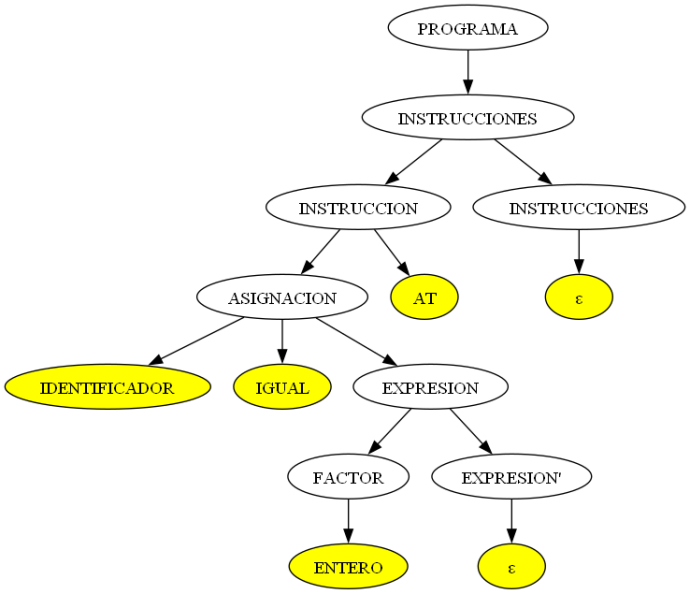

Figura 1: Árbol Sintáctico Generado

Listing 7: Ejmplo 02

```
1       #Asignacion de funcion
2       def hola(){
3       }
```

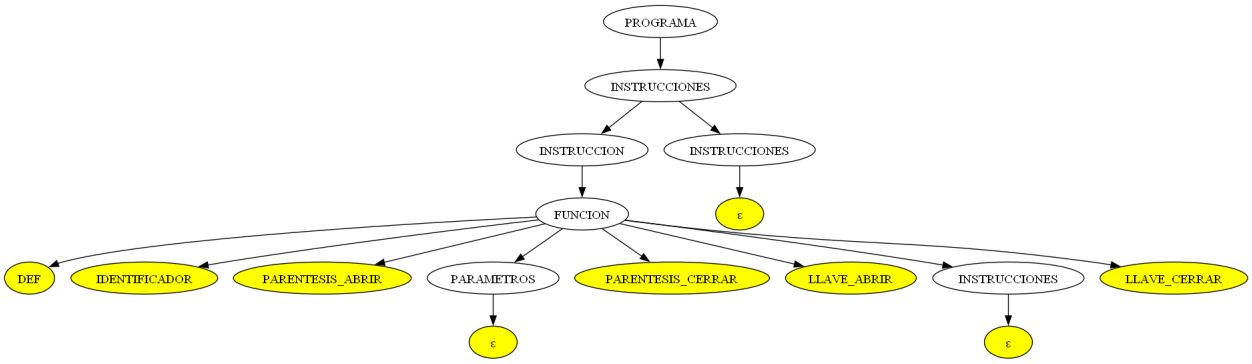

Figura 2: Árbol Sintáctico Generado

Los demás ejemplos estarán en la carpeta ejemplos dentro del repositorio en GitHub.