“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

CARRERA: INGENIERÍA DE SOFTWARE
WAYRASIMI: IMPLEMENTACIÓN DE SU ANALIZADOR LÉXICO
COMPILADORES

ESTUDIANTES:

Ortiz Castañeda Jorge Luis
Huamani Huamani Jhordan Steven Octavio
Flores Leon Miguel Angel

DOCENTE:
Vicente Enrique Machaca Arceda

Arequipa, Perú
23 de marzo de 2025

Índice

1. Introducción

En el presente informe, se expondrá el desarrollo de nuestro lenguaje en quechua, denominado "WayraSimi", durante su fase de análisis y pruebas con el analizador léxico. Para llevar a cabo esta implementación, se ha utilizado el lenguaje de programación Python, junto con la librería "ply.lex", con el objetivo de construir un compilador básico para nuestro lenguaje propuesto.

2. WayraSimi

Presentamos WayraSimi, un lenguaje de programación compilado que fusiona la claridad de Python con la eficiencia de Go. Diseñado para aplicaciones de alto rendimiento y sistemas concurrentes, WayraSimi busca ser una herramienta poderosa y accesible para desarrolladores.

- Ejemplo de documento: WayraSimi.ws
- Composición: Wayra (Viento, Aire) + Simi (Palabra, Lenguaje)
- Significado: Lenguaje del Viento o Lenguaje Veloz, implicando rapidez y eficiencia.
- Gophy: Mascota del Lenguaje

3. Justificación y Descripción del Lenguaje

3.1. Sintaxis Python-inspirada

Utiliza la indentación para definir bloques de código, buscando la legibilidad y simplicidad sintáctica de Python.

3.2. Tipado estático e inferencia de tipos

Similar a Go, WayraSimi será un lenguaje de tipado estático para garantizar la seguridad y el rendimiento.

3.3. Concurrencia integrada

Inspirado en Go, WayraSimi tendrá soporte nativo para concurrencia ligera (gorutinas) y comunicación entre procesos (canales).

4. Tipos de Datos en WayraSimi

- **Yupay (entero)**: Números enteros, sin parte decimal (int, int32, int64).
- **Chiqi_kay (flotante)**: Números con parte decimal (punto flotante, float32, float64).
- **Qillga (texto)**: Cadenas de caracteres, texto (string).

WayraSimi soporta varios tipos de datos, incluyendo enteros, flotantes, texto, booleanos, listas y mapas, cada uno con su propia representación y uso.

5. Ejemplos

```
ruray hatunRuray() {
    imprimiy(" Allin  punchaw, Pachamama!");
}
```

Listing 1: Hola Mundo

```
ruray hatunRuray() {
    para i := 0; i < 5; i++ {
        para j := 0; j < 5; j++ {
            imprimiy(i, j);
        }
    }
}
```

Listing 2: Bucles Anidados

```
ruray factorial(n yupay) yupay {
    sichus n == 0 {
        kutipay 1;
    }
    kutipay n * factorial(n-1);
}

ruray hatunRuray() {
    imprimiy(factorial(5));
}
```

Listing 3: Recursividad

```
sichus (wata > 18) {
    imprimiy("Eres mayor de edad");
} mana sichus (wata == 18) {
    imprimiy("Eres justo mayor de edad");
} mana {
    imprimiy("Eres menor de edad");
}
```

Listing 4: Si anidado

```
# Este es un comentario de línea

ruray hatun_ruray(){
    imprimiy("Este no es un comentario")
    imprimiy(1.4)
}
```

Listing 5: Comentario

6. Analizador Léxico en Python

6.1. pruebita.py

```

import ply.lex as lex
from prettytable import PrettyTable


tokens = [
    'YUPAY_TOKEN',
    'CHIQI_KAY_TOKEN',
    'QILLQA_TOKEN',
    'CHIQAP_TOKEN',
    'IDENTIFICADOR_TOKEN',
    'OPERADOR_MAS',
    'OPERADOR_MENOS',
```

```
    'OPERADOR_PACHA',
    'OPERADOR_RAKI',
    'OPERADOR_MODULO',
    'OPERADOR_ASIGNACION',
    'OPERADOR_IGUALDAD',
    'OPERADOR_MANA_IGUAL',
    'OPERADOR_MENOR',
    'OPERADOR_MAYOR',
    'OPERADOR_MENOR_IGUAL',
    'OPERADOR_MAYOR_IGUAL',
    'OPERADOR_LOGICO_WAN',
    'OPERADOR_LOGICO_UTAQ',
    'OPERADOR_LOGICO_MANA',
    'PARENTESIS_ABRE',
    'PARENTESIS_CIERRA',
    'LLAVE_ABRE',
    'LLAVE_CIERRA',
    'CORCHETE_ABRE',
    'CORCHETE_CIERRA',
    'COMA_TOKEN',
    'PUNTO_TOKEN',
    'DOS_PUNTOS_TOKEN',
    'PUNTO_Y_COMA_TOKEN',
    'COMENTARIO_TOKEN_LINEA',
    'COMENTARIO_TOKEN_BLOQUE_ABRE',
    'COMENTARIO_TOKEN_BLOQUE_CIERRA',
    'PALABRA_RESERVADA_SICHUS',
    'PALABRA_RESERVADA_MANA_SICHUS',
    'PALABRA_RESERVADA_MANA',
    'PALABRA_RESERVADA_PARA',
    'PALABRA_RESERVADA_RURAY',
    'PALABRA_RESERVADA_KUTIPAY',
    'PALABRA_RESERVADA_IMPRIMIY',
    'PALABRA_RESERVADA_AYLLU',
    'PALABRA_RESERVADA_VAR',
    'PALABRA_RESERVADA_TAKYAQ',
    'PALABRA_RESERVADA_UYWA',
    'PALABRA_RESERVADA_UYA',
    'HATUN_RURAY_TOKEN',
]

reserved_words = {
    'sichus': 'PALABRA_RESERVADA_SICHUS',
    'mana sichus': 'PALABRA_RESERVADA_MANA_SICHUS',
    'mana': 'PALABRA_RESERVADA_MANA',
    'para': 'PALABRA_RESERVADA_PARA',
    'ruray': 'PALABRA_RESERVADA_RURAY',
    'kutipay': 'PALABRA_RESERVADA_KUTIPAY',
    'imprimiy': 'PALABRA_RESERVADA_IMPRIMIY',
    'ayllu': 'PALABRA_RESERVADA_AYLLU',
    'variable': 'PALABRA_RESERVADA_VAR',
    'takyaq': 'PALABRA_RESERVADA_TAKYAQ',
    'uywa': 'PALABRA_RESERVADA_UYWA',
    'uya': 'PALABRA_RESERVADA_UYA',
    'chiqap': 'CHIQAP_TOKEN',
    'mana_chiqap': 'CHIQAP_TOKEN',
    'wan': 'OPERADOR_LOGICO_WAN',
    'utaq': 'OPERADOR_LOGICO_UTAQ'
}

t_OPERADOR_MAS    = r'\+'
t_OPERADOR_MENOS   = r'-'
t_OPERADOR_PACHA   = r'\*'
t_OPERADOR_RAKI    = r'/'
t_OPERADOR_MODULO  = r'%'
t_OPERADOR_ASIGNACION = r'='
t_OPERADOR_IGUALDAD = r'=='
t_OPERADOR_MANA_IGUAL = r'!='
t_OPERADOR_MENOR   = r'<'
t_OPERADOR_MAYOR   = r'>'
t_OPERADOR_MENOR_IGUAL = r'<='
t_OPERADOR_MAYOR_IGUAL = r'>='
t_PARENTESIS_ABRE  = r'\('
```

```
t_PARENTESIS_CIERRA = r'\)'
t_LLAVE_ABRE        = r'\{'
t_LLAVE_CIERRA       = r'\}'
t_CORCHETE_ABRE     = r'\['
t_CORCHETE_CIERRA    = r'\]'
t_COMA_TOKEN        = r','
t_PUNTO_TOKEN       = r'\.'
t_DOS_PUNTOS_TOKEN = r':'
t_PUNTO_Y_COMA_TOKEN = r';'

def t_CHIQI_KAY_TOKEN(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_YUPAY_TOKEN(t):
    r'\d+'
    t.value = int(t.value)
    return t


def t_QILLQA_TOKEN(t):
    r'("([^"]*)")|(\'([^\']*)\')'
    t.value = t.value[1:-1]
    return t

def t_HATUN_RURAY_TOKEN(t):
    r'hatun_ruray'
    return t

def t_IDENTIFICADOR_TOKEN(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    if t.value in reserved_words:
        t.type = reserved_words[t.value]
    return t

def t_COMENTARIO_TOKEN_LINEA(t):
    r'\#.*'
    pass

def t_COMENTARIO_TOKEN_BLOQUE_ABRE(t):
    r'/\*'
    print(">> COMENTARIO_TOKEN_BLOQUE_ABRE encontrado")
    t.lexer.comment_start = t.lexer.lexpos
    t.lexer.level = 1
    t.lexer.begin('comment')

def t_comment_COMENTARIO_TOKEN_BLOQUE_CIERRA(t):
    r'\*/'
    print(">> COMENTARIO_TOKEN_BLOQUE_CIERRA encontrado")
    t.lexer.level -= 1
    if t.lexer.level == 0:
        t.lexer.begin('INITIAL')

def t_comment_error(t):
    print(f">> ERROR en comentario de bloque en pos {t.lexer.comment_start}:
        ↪ '{t.value[0]}'")
    print(f"Estado Actual: {t.lexer.current_state()}")
    t.lexer.skip(1)

def t_comment_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_comment_ignore = r'.'


states = (
   ('comment','exclusive'),
)

t_ignore            = ' \t\r'

```

```python
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_ANY_error(t):
    print(f"Carácter alterno '{t.value[0]}' en la línea {t.lineno}, posición
        ↪ {t.lexpos}, estado: {t.lexer.current_state()}")
    t.lexer.skip(1)

lexer = lex.lex()

def analyze_file(filepath):
    try:
        with open(filepath, 'r', encoding='utf-8') as file:
            data = file.read()
    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{filepath}'.")
        return

    lexer.input(data)
    tokens_list = []

    table = PrettyTable(["Tipo", "Valor", "Línea", "Posición"])

    while True:
        tok = lexer.token()
        if not tok:
            break
        tokens_list.append(tok)
        table.add_row([tok.type, tok.value, tok.lineno, tok.lexpos])
    print(table)
    return tokens_list

if __name__ == '__main__':
    example_files = [
        "ejemplito1.txt",
        "ejemplito2.txt",
        "ejemplito3.txt",
        "ejemplito4.txt",
        "ejemplito5.txt"
    ]

    for filepath in example_files:
        print(f"\n--- Analizando el archivo: {filepath} ---")
        tokens = analyze_file(filepath)
        if tokens:
            print(f"Se analizaron {len(tokens)} tokens en el archivo '{filepath}'.")
```

Listing 6: Lexema en lenguaje Python

6.2. Explicación pruebita.py

6.2.1. Importación de la librería

Se importa la librería **ply.lex**, que es una herramienta para construir analizadores léxicos en Python.

6.2.2. Definición de los tokens

Se define una lista de tokens (**'YUPAY_TOKEN'**, **'CHIQAP_TOKEN'**, etc.), que son los componentes léxicos que el analizador reconocerá. Cada token representa un elemento del lenguaje, como números, operadores, palabras reservadas, etc.

6.2.3. Expresiones regulares para tokens simples

Aquí se definen las expresiones regulares para los tokens simples, como operadores (+, -, *, etc.), símbolos de puntuación (",", ";", ":", etc.) y palabras reservadas (sichus, mana, ruray, etc.).

6.2.4. Funciones para tokens complejos

Algunos tokens requieren lógica adicional para ser reconocidos. Por ejemplo:

- **Números (YUPAY_TOKEN):** Reconoce números enteros o decimales y los convierte a int o float.
- **Booleanos (CHIQAP_TOKEN):** Reconoce los valores booleanos chiqap (verdadero) y mana_chiqap (falso).
- **Cadenas de texto (QILLQA_TOKEN):**Reconoce cadenas de texto entre comillas simples o dobles.
- **Identificadores (IDENTIFICADOR_TOKEN):**Reconoce nombres de variables o funciones.

6.2.5. Manejo de comentarios y espacios

- Comentarios de línea
- Espacios y tabulaciones
- Saltos de línea

6.2.6. Manejo de errores

Si se encuentra un carácter no reconocido, se imprime un mensaje de error y se ignora el carácter.

6.2.7. Creación del lexer

Se crea una instancia del analizador léxico.

6.2.8. Pruebas y análisis de archivos

- **Función test_lexer:** Prueba el lexer con una cadena de entrada.
- **Función analyze_file:** Lee un archivo y analiza su contenido léxicamente.

6.3. Ejecución principal

Se analizan varios archivos de ejemplo y se imprimen los tokens encontrados.

6.4. Output de Pruebita.py

```
--- Analizando el archivo: ejemplito1.txt ---
+---------------------------+---------------------------+-------+----------+
|           Tipo            |          Valor            | Línea | Posición |
+---------------------------+---------------------------+-------+----------+
|  PALABRA_RESERVADA_RURAY  |          ruray            |   1   |    0     |
|     HATUN_RURAY_TOKEN     |       hatun_ruray         |   1   |    6     |
|      PARENTESIS_ABRE      |             (             |   1   |    17    |
|     PARENTESIS_CIERRA     |             )             |   1   |    18    |
|         LLAVE_ABRE        |             {             |   1   |    20    |
| PALABRA_RESERVADA_IMPRIMIY |         imprimiy          |   2   |    26    |
|      PARENTESIS_ABRE      |             (             |   2   |    34    |
|        QILLQA_TOKEN       | ¡Allin punchaw, Pachamama! |   2   |    35    |
|     PARENTESIS_CIERRA     |             )             |   2   |    63    |
|     PUNTO_Y_COMA_TOKEN    |             ;             |   2   |    64    |
|        LLAVE_CIERRA       |             }             |   3   |    66    |
+---------------------------+---------------------------+-------+----------+
Se analizaron 11 tokens en el archivo 'ejemplito1.txt'.
```

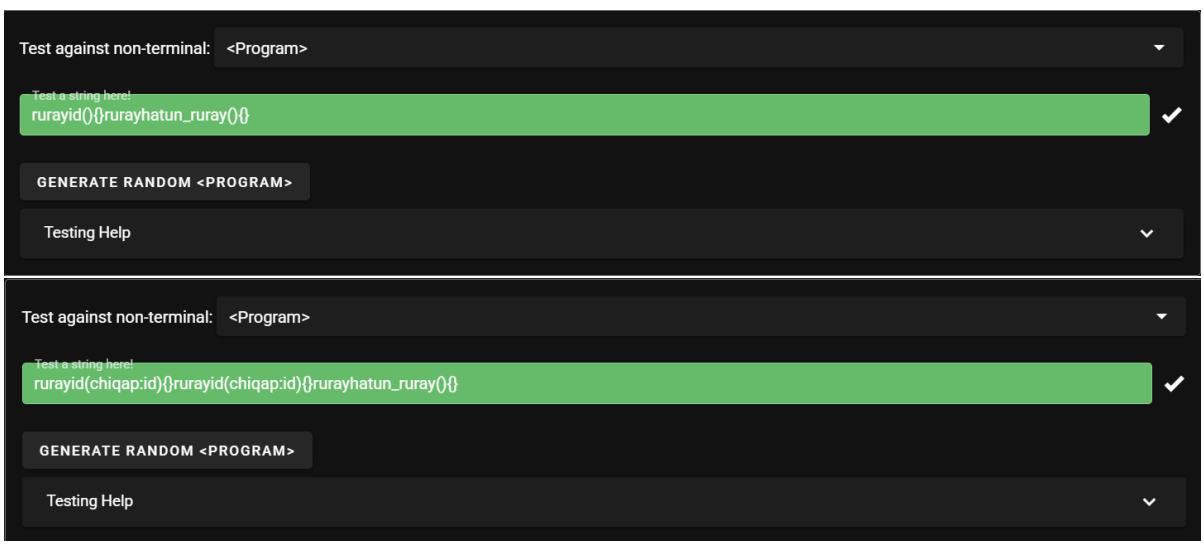
7. Gramática en BNF

```
<Program> ::= <DefinitionList> <Principal>

<DefinitionList> ::= <FunctionDef> <DefinitionList> | E

<FunctionDef> ::= "ruray" <IDENTIFICADOR_TOKEN> "(" <ParamListOpt> ")" <TypeOpt>
    ↪ <Block>

<Principal> ::= "ruray" "hatun_ruray" "(" ")" <Block>

<ParamListOpt> ::= <ParamList> | E
<ParamList> ::= <Param> <MoreParams>
<MoreParams> ::= "," <Param> <MoreParams> | E
<Param> ::= <Type> ":" <IDENTIFICADOR_TOKEN>

<Type> ::= "yupay" | "chiqi" | "chiqap" | "qillqa"
<TypeOpt> ::= <Type> | E

<Block> ::= "{" <Instrucciones> "}"
<Instrucciones> ::= <Instruccion> <Instrucciones> | E

<Instruccion> ::= <DeclaracionVariables>
                | <PrintStmt>
                | <Bucle>
                | <Estructura_If>
                | <Retorno>
                | <incrementos> ";"

<DeclaracionVariables> ::= "var" <ListaIdentificadores> <Type> <InicializacionOpt> ";"
                         | <ListaIdentificadores> ":=" <opciones> ";"
                         | <IDENTIFICADOR_TOKEN> ":=" <opciones> ";"

<ListaIdentificadores> ::= <IDENTIFICADOR_TOKEN> <MasIdentificadores>
<MasIdentificadores> ::= "," <IDENTIFICADOR_TOKEN> <MasIdentificadores> | E
<InicializacionOpt> ::= "=" <opciones> | E

<PrintStmt> ::= "imprimiy" "(" <ArgumentPrint> ")" ";"
<ArgumentPrint> ::= <opciones> <MasArgumentosPrint> | E
<MasArgumentosPrint> ::= "," <opciones> <MasArgumentosPrint> | E

<opciones> ::= <funcion>
             | <comparacion>
             | <IDENTIFICADOR_TOKEN>
             | <datos>
             | <operaciones_matematicas>
             | <incrementos>

<incrementos> ::= <IDENTIFICADOR_TOKEN> "++" | <IDENTIFICADOR_TOKEN> "--"

<operaciones_matematicas> ::= <termino> <expresion_tail>
<expresion_tail> ::= "+" <termino> <expresion_tail>
                   | "-" <termino> <expresion_tail>
                   | E
<termino> ::= <factor> <termino_tail>
<termino_tail> ::= "*" <factor> <termino_tail>
                 | "/" <factor> <termino_tail>
                 | "%" <factor> <termino_tail>
                 | E

<factor> ::= <unidad> | "-" <unidad>
<unidad> ::= <valor> | "(" <operaciones_matematicas> ")"
<valor> ::= <YUPAY_TOKEN> | <CHIQI_TOKEN> | <CHIQAP_TOKEN> | <QILLQA_TOKEN> |
    ↪ <IDENTIFICADOR_TOKEN> | <funcion>

<funcion> ::= <IDENTIFICADOR_TOKEN> "(" <argumentosFuncion> ")"
<argumentosFuncion> ::= <opciones> <MasArgumentosFuncion> | E
<MasArgumentosFuncion> ::= "," <opciones> <MasArgumentosFuncion> | E

<comparacion> ::= <logical_expression>
<logical_expression> ::= "mana" "(" <expression> ")"
                       | "(" <expression> ")"
```

```
                        | <expression>

<expression> ::= <compa> <expression_conti>
<expression_conti> ::= <opciones_operador_logico> <expression> | E
<compa> ::= <opciones> <opciones_operador_comparacion> <opciones>
          | <opciones> <opciones_operador_logico> <opciones>

<Bucle> ::= <for> | <while> | <bucle_infinito>
<for> ::= "para" "(" <LoopInitialization> ";" <opciones> ";" <LoopUpdate> ")" <Block>
<while> ::= "para" "(" <opciones> ")" <Block>
<bucle_infinito> ::= "para" <Block>

<LoopInitialization> ::= <IDENTIFICADOR_TOKEN> ":=" <asignacion_bucle>
<asignacion_bucle> ::= <funcion> | <IDENTIFICADOR_TOKEN> | <datos>
<LoopUpdate> ::= <operaciones_matematicas>

<Estructura_If> ::= "sichus" "(" <opciones> ")" <Block> <Else_opcional>
<Else_opcional> ::= "mana_sichus" <argumentosELseIf> <Block> | E
<argumentosELseIf> ::= "(" <opciones> ")" | E

<Retorno> ::= "kutipay" <IDENTIFICADOR_TOKEN> ";"
            | "kutipay" ";"
            | "kutipay" <datos> ";"
            | "pakiy" ";"

<datos> ::= <YUPAY_TOKEN> | <CHIQI_TOKEN> | <CHIQAP_TOKEN> | <QILLQA_TOKEN> |
    ↪ <operaciones_matematicas> | <IDENTIFICADOR_TOKEN>

<YUPAY_TOKEN> ::= "num"
<CHIQI_TOKEN> ::= "num" "." "num"
<CHIQAP_TOKEN> ::= "chiqaq" | "mana_chiqaq"
<QILLQA_TOKEN> ::= "texto"
<IDENTIFICADOR_TOKEN> ::= "id"

<OPERADOR_ASIGNACION> ::= ":="
<OPERADOR_LOGICO_UTAQ> ::= "utaq"
<OPERADOR_LOGICO_WAN> ::= "wan"
<OPERADOR_LOGICO_MANA> ::= "mana"
<opciones_operador_logico> ::= "wan" | "utaq"
<opciones_operador_comparacion> ::= "==" | "!=" | "<" | ">" | "<=" | ">="
```

Listing 7: Gramática

7.1. Output del BNF



8. FIRST and FOLLOW

```python
import re
import csv

class GramaticaAnalyzer:
    def __init__(self):
        self.producciones = {}
        self.first = {}
        self.follow = {}
        self.terminales = set()
        self.no_terminales = set()

    def leer_gramatica(self, archivo):
        with open(archivo, 'r', encoding='utf-8') as f:
            contenido = f.read()

        # Extraer las reglas de producción
        lineas = contenido.split('\n')
        produccion_actual = None

        for linea in lineas:
            if '::=' in linea:
                # Nueva producción
                izquierda, derecha = linea.split('::=')
                izquierda = izquierda.strip()
                if izquierda.startswith('<') and izquierda.endswith('>'):
                    izquierda = izquierda[1:-1]  # Quitar < >
                    produccion_actual = izquierda
                    self.no_terminales.add(produccion_actual)
                    self.producciones[produccion_actual] = []

                    # Procesar la parte derecha
                    self._procesar_produccion(derecha, produccion_actual)

            elif '|' in linea and produccion_actual:
                # Producción alternativa
                self._procesar_produccion(linea, produccion_actual)

    def _procesar_produccion(self, texto, produccion_actual):
        # Limpiar y dividir la producción
        partes = texto.split('|')
        for parte in partes:
            parte = parte.strip()
            if parte:
                simbolos = []
                # Encontrar símbolos entre < > o tokens/terminales
                tokens = re.findall(r'<[^>]+>|"[^"]+"|[A-Z_]+_TOKEN', parte)
                for token in tokens:
                    if token.startswith('<') and token.endswith('>'):
                        simbolo = token[1:-1]  # Quitar < >
                        self.no_terminales.add(simbolo)
                    else:
                        simbolo = token.strip('"')
                        self.terminales.add(simbolo)
                    simbolos.append(simbolo)

                if simbolos:
                    self.producciones[produccion_actual].append(simbolos)
                elif 'E' in parte:  # Epsilon
                    self.producciones[produccion_actual].append(['epsilon'])
                    self.terminales.add('epsilon')

    def calcular_first(self):
        self.first = {nt: set() for nt in self.no_terminales}
        self.first.update({t: {t} for t in self.terminales})

        while True:
            cambios = False
            for nt in self.no_terminales:
                for produccion in self.producciones[nt]:
                    first_anterior = len(self.first[nt])

                    # Para cada símbolo en la producción
```

```
                    todo_epsilon = True
                    for simbolo in produccion:
                        if simbolo == 'epsilon':
                            self.first[nt].add('epsilon')
                            break

                        if simbolo in self.first:
                            self.first[nt].update(self.first[simbolo] - {'epsilon'})
                            if 'epsilon' not in self.first[simbolo]:
                                todo_epsilon = False
                                break
                        else:
                            self.first[nt].add(simbolo)
                            todo_epsilon = False
                            break

                    if todo_epsilon and len(produccion) > 0:
                        self.first[nt].add('epsilon')

                    if len(self.first[nt]) > first_anterior:
                        cambios = True

            if not cambios:
                break

    def calcular_follow(self):
        self.follow = {nt: set() for nt in self.no_terminales}
        self.follow['Program'].add('$')  # Símbolo inicial

        while True:
            cambios = False
            for nt in self.no_terminales:
                for produccion in self.producciones[nt]:
                    for i, simbolo in enumerate(produccion):
                        if simbolo in self.no_terminales:
                            follow_anterior = len(self.follow[simbolo])

                            # Si es el último símbolo
                            if i == len(produccion) - 1:
                                self.follow[simbolo].update(self.follow[nt])
                            else:
                                # Calcular FIRST del resto de la producción
                                siguiente = produccion[i + 1]
                                if siguiente in self.first:
                                    self.follow[simbolo].update(self.first[siguiente]
                                        ↪ - {'epsilon'})
                                    if 'epsilon' in self.first[siguiente]:
                                        self.follow[simbolo].update(self.follow[nt])
                                else:
                                    self.follow[simbolo].add(siguiente)

                            if len(self.follow[simbolo]) > follow_anterior:
                                cambios = True

            if not cambios:
                break

    def guardar_csv(self, archivo_salida):
        with open(archivo_salida, 'w', newline='', encoding='utf-8') as f:
            writer = csv.writer(f)
            writer.writerow(['No Terminal', 'FIRST', 'FOLLOW'])

            for nt in sorted(self.no_terminales):
                first_str = ', '.join(sorted(self.first[nt]))
                follow_str = ', '.join(sorted(self.follow[nt]))
                writer.writerow([nt, first_str, follow_str])

def main():
    analizador = GramaticaAnalyzer()
    analizador.leer_gramatica('GramaticaBNF.txt')
    analizador.calcular_first()
    analizador.calcular_follow()
    analizador.guardar_csv('primerosYsiguientes.csv')
```

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

Listing 8: Gramática

8.1. Explicación del código: cálculo de FIRST y FOLLOW

FIRST y **FOLLOW** de cada no terminal, y los exporta a un archivo CSV.

8.2. Tabla desde CSV

- **import re**, **import csv**: Se importan las bibliotecas necesarias. **re** se usa para trabajar con expresiones regulares y **csv** para exportar los resultados.
- **class GramaticaAnalyzer:** Se define la clase principal que encapsula toda la funcionalidad.
- **__init__**: Se inicializan los atributos: diccionarios para producciones, conjuntos FIRST y FOLLOW, y conjuntos para terminales y no terminales.
- **leer_gramatica(archivo)**: Método para leer un archivo BNF. Divide el contenido en líneas y procesa cada producción. Si encuentra una línea con **::=**, se trata de una nueva producción; si contiene |, se interpreta como una producción alternativa.
- **_procesar_produccion**: Método auxiliar que analiza una parte derecha de una producción. Usa expresiones regulares para extraer símbolos entre **<>**, literales entre comillas, o tokens escritos en mayúsculas. Clasifica los símbolos como terminales o no terminales.
- **calcular_first**: Calcula el conjunto FIRST de cada no terminal. Se inicializan los conjuntos y se aplican las reglas clásicas de cálculo:
  - Si la producción comienza con un terminal, se añade al conjunto.
  - Si el símbolo puede derivar **epsilon**, se sigue analizando el siguiente símbolo.
  - El proceso se repite hasta que no hay más cambios.
- **calcular_follow**: Calcula el conjunto FOLLOW.
  - Inicializa todos los conjuntos y agrega **$** al FOLLOW del símbolo inicial (**Program**).
  - Para cada aparición de un no terminal dentro de una producción, se calculan los posibles siguientes símbolos y se aplican las reglas para FOLLOW, incluyendo casos donde el símbolo es el último o seguido por **epsilon**.
- **guardar_csv**: Escribe en un archivo CSV los resultados de FIRST y FOLLOW para cada no terminal.
- **main**(): Crea una instancia del analizador, lee el archivo
- **GramaticaBNF.txt**, calcula FIRST y FOLLOW, y guarda los resultados en
- **primerosYsiguientes.csv**.
- **if __name__ == '__main__'**: Ejecuta la función
- **main** cuando el archivo se ejecuta como script principal.

9. Tabla de Transiciones

```
import csv

def leer_gramatica(archivo):
    with open(archivo, 'r') as f:
        lineas = f.readlines()
    # Filtrar líneas vacías y comentarios
    return [linea.strip() for linea in lineas if linea.strip() and '::=' in linea]
```

```python

def obtener_no_terminales(gramatica):
    return list({linea.split('::=')[0].strip()[1:-1] for linea in gramatica})

def obtener_terminales(gramatica):
    terminales = set()
    for linea in gramatica:
        if '::=' not in linea:
            continue
        produccion = linea.split('::=')[1]
        # Buscar elementos entre comillas
        partes = produccion.split('"')
        for i in range(1, len(partes), 2):
            if partes[i]:
                terminales.add(partes[i])
    return sorted(list(terminales))

def procesar_produccion(produccion):
    # Dividir por | para obtener alternativas
    alternativas = produccion.split('|')
    resultado = []

    for alt in alternativas:
        alt = alt.strip()
        if alt == 'E':
            resultado.append(['epsilon'])
        else:
            elementos = []
            partes = alt.split('"')
            for i, parte in enumerate(partes):
                parte = parte.strip()
                if i % 2 == 0:  # No está entre comillas
                    # Buscar no terminales
                    tokens = [t.strip() for t in parte.split() if t.strip()]
                    for token in tokens:
                        if token.startswith('<') and token.endswith('>'):
                            elementos.append(token[1:-1])
                else:  # Está entre comillas
                    elementos.append(parte)
            resultado.append(elementos)

    return resultado

def crear_tabla_transiciones(gramatica):
    no_terminales = obtener_no_terminales(gramatica)
    terminales = obtener_terminales(gramatica)

    # Crear diccionario para la tabla
    tabla = {nt: {t: '' for t in terminales + ['$']} for nt in no_terminales}

    # Procesar cada regla
    for linea in gramatica:
        no_terminal, produccion = [p.strip() for p in linea.split('::=')]
        no_terminal = no_terminal[1:-1]  # Quitar < >

        alternativas = procesar_produccion(produccion)

        for alt in alternativas:
            if alt == ['epsilon']:
                tabla[no_terminal]['$'] = 'epsilon'
                for t in terminales:
                    if not tabla[no_terminal][t]:
                        tabla[no_terminal][t] = 'epsilon'
            else:
                # Si el primer elemento es terminal
                if alt[0] in terminales:
                    tabla[no_terminal][alt[0]] = ' '.join(alt)
                else:
                    # Si el primer elemento es no terminal
                    tabla[no_terminal][terminales[0]] = ' '.join(alt)

    return tabla, terminales, no_terminales

```

```python
def guardar_csv(tabla, terminales, no_terminales, archivo_salida):
    with open(archivo_salida, 'w', newline='') as f:
        writer = csv.writer(f)
        # Escribir encabezados
        writer.writerow([''] + terminales + ['$'])
        # Escribir filas
        for nt in no_terminales:
            fila = [nt]
            for t in terminales + ['$']:
                fila.append(tabla[nt][t])
            writer.writerow(fila)

def main(archivo_entrada, archivo_salida):
    # Leer la gramática
    gramatica = leer_gramatica(archivo_entrada)

    # Crear la tabla de transiciones
    tabla, terminales, no_terminales = crear_tabla_transiciones(gramatica)

    # Guardar la tabla en CSV
    guardar_csv(tabla, terminales, no_terminales, archivo_salida)

if __name__ == '__main__':
    archivo_entrada = "GramaticaBNF.txt"
    archivo_salida = "tablaTransiciones.csv"
    main(archivo_entrada, archivo_salida)
```

Listing 9: Gramática

9.1. Explicación del código: Tabla de Transiciones

- **import csv**: Se importa el módulo `csv` para manejar la escritura de archivos en formato CSV.
- **leer_gramatica(archivo)**: Abre y lee el archivo de la gramática, filtrando líneas vacías y aquellas que no contienen la cadena `::=`. Devuelve una lista con las producciones válidas.
- **obtener_no_terminales(gramatica)**: Extrae los no terminales desde la parte izquierda de las producciones, eliminando los signos <>. Devuelve una lista sin duplicados.
- **obtener_terminales(gramatica)**: Recorre la parte derecha de cada producción buscando terminales encerrados entre comillas. Los almacena en un conjunto para evitar duplicados y devuelve la lista ordenada.
- **procesar_produccion(produccion)**: Dada la parte derecha de una producción, divide las alternativas separadas por `|`. Cada alternativa se analiza en partes entre comillas para distinguir terminales de no terminales. El resultado es una lista de listas, donde cada sublista representa una alternativa de producción.
- **crear_tabla_transiciones(gramatica)**:
  - **no_terminales** y **terminales** se obtienen usando las funciones anteriores.
  - Se crea una tabla (diccionario anidado) inicializando cada celda vacía para todas las combinaciones de no terminales y terminales más `$`.
  - Para cada producción, se procesan sus alternativas. Si una alternativa es `epsilon`, se llena la fila con `epsilon` en las columnas vacías. Si la alternativa empieza con un terminal, se asigna directamente. Si empieza con un no terminal, se asigna arbitrariamente en la primera columna terminal (esto debería mejorarse usando `FIRST`).
- **guardar_csv(tabla, terminales, no_terminales, archivo_salida)**: Crea un archivo CSV con la tabla. La primera fila contiene los terminales y `$`. Luego, cada fila corresponde a un no terminal, seguido de sus transiciones.
- **main(archivo_entrada, archivo_salida)**: Ejecuta el proceso completo: lee la gramática, crea la tabla de transiciones y la guarda en un archivo.
- **if __name__ == '__main__'**: Indica que el bloque solo debe ejecutarse cuando el archivo se corre directamente. Define las rutas de entrada y salida, y llama a `main`.