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. Consideraciones iniciales

Se debe elaborar un informe con la descripción del trabajo. El informe puede estar desarrollado en LaTeX.

2. Descripción del trabajo

2.1. Implementación

Desarrollar un programa compilador, este deberá tener como entrada y salida:

- **Entrada:** Código fuente en el lenguaje propuesto.
- **Salida:** Código en lenguaje ensamblador (SPIM).

El programa debe integrar los módulos desarrollados anteriormente como el analizador léxico, analizador sintáctico y analizador semántico. Además, debe mostrar si hay errores de código en cada una de estas fases.

2.2. 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 está bien, puede utilizar esta herramienta (la gramática no debe ser ambigua y debe estar factorizada por la izquierda).
- **Implementación:** Enlace al repositorio.
- **Conclusiones.**

3. 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.

4. URL de Repositorio Github

- URL del Repositorio GitHub para clonar o recuperar.
- `https://github.com/CDanielAg/Final_Compiladores.git?authuser=1`

5. Actividades con el repositorio GitHub

5.1. 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
```

6. 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.

7. 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.

8. 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 -> RETORNAR EXPRESION AT
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 -> EXPRESION PARAMETROS'
PARAMETROS' -> COMA EXPRESION 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' -> MASEXPRESION
EXPRESION' -> COMPARACION FACTOR
EXPRESION' -> ''

MASEXPRESION -> OPERADOR FACTOR MASEXPRESION
MASEXPRESION -> ''

OPERADOR -> MAS
OPERADOR -> MENOS
OPERADOR -> ENTRE
```

```
OPERADOR -> POR

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

FACTOR -> IDENTIFICADOR INVFUNC
FACTOR -> ENTERO
FACTOR -> FLOTANTE
FACTOR -> BOOLEANO
FACTOR -> CADENA
INVFUNC -> PARENTESIS_ABRIR PARAMETROS PARENTESIS_CERRAR
INVFUNC -> ''

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

9. 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$_sintactica.py$

```python
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
```

```
144             rules = read_grammar(grammar_file)
145             alphabet, nonterminals, terminals = collect_alphabet_and_nonterminals(rules)
146             firsts = collect_firsts(rules, nonterminals, terminals)
147             follows = collect_follows(rules, nonterminals, firsts)
148             # Generar la tabla LL(1) y escribirla en un archivo CSV
149             rule_table = make_rule_table(rules, nonterminals, terminals, firsts, follows)
150             write_csv(rule_table, output_file)
151             # Escribir los no terminales en un archivo de texto
152             write_nonterminals(nonterminals, nonterminals_file)
153             print(f"Tabla LL(1) generada y guardada en {output_file}")
154             print(f"No terminales guardados en {nonterminals_file}")
155 
156         if __name__ == '__main__':
157             main()
```

10. 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

```
1         import csv
2         import re
3 
4         EPSILON = "''"
5 
6         class LLParser:
7             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:
```

```
                self.rules = [line.strip() for line in f if '->' in line]

        def _load_tokens(self, tokens_file):
            with open(tokens_file, 'r') as f:
                self.tokens = f.read().strip().split()

        def _collect_alphabet_and_symbols(self):
            for rule in self.rules:
                lhs, rhs = rule.split('->')
                nonterminal = lhs.strip()
                development = rhs.strip().split()

                if nonterminal not in self.nonterminals:
                    self.nonterminals.append(nonterminal)

                for symbol in development:
                    if symbol != EPSILON:
                        if symbol not in self.alphabet:
                            self.alphabet.append(symbol)

            self.terminals = [symbol for symbol in self.alphabet if symbol not in self.nonterminals]

        def _load_rule_table(self):
            # Load the rule table from an external CSV file
            with open('ll1_table.csv', mode='r') as file:
                csv_reader = csv.DictReader(file)
                for row in csv_reader:
                    nonterminal = row['Nonterminal']
                    self.rule_table[nonterminal] = {}
                    for terminal, rule in row.items():
                        if terminal != 'Nonterminal' and rule:
                            self.rule_table[nonterminal][terminal] = rule

        def parse_input(self):
            stack = ['$', self.nonterminals[0]]
            index = 0
            input_tokens = self.tokens + ['$']

            rows = []
            while len(stack) > 0:
                top = stack.pop()
                current_token = input_tokens[index]
                rule = self.rule_table.get(top, {}).get(current_token) if top in self.nonterminals
                     else ""

                rows.append([" ".join(stack), " ".join(input_tokens[index:]), f"{top} ->
                     {rule.split('->')[1].strip()}" if rule else "Accept" if top == '$' and
                     current_token == '$' else ""])

                if top == current_token:
                    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()
```

11. 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
```

```
20
21      def generar_arbol_sintactico(rastreo, nombre_archivo):
22          dot = Digraph(comment='rbol Sintctico')
23          contador_nodos = 0
24          nodos = {}
25          reglas_diccionario = {}
26
27          # Crear un nodo raz para comenzar el rbol
28          raiz = None
29
30          # Crear nodos usando las reglas de la traza
31          for entrada in rastreo:
32              regla = entrada['Rule']
33              if '->' in regla:
34                  cabeza, produccion = regla.split('->')
35                  cabeza = cabeza.strip()
36                  simbolos_produccion = [simbolo for simbolo in produccion.strip().split() if simbolo
                        != "'"]
37
38                  # Almacenar la regla en el diccionario sin sobrescribir reglas existentes
39                  if cabeza in reglas_diccionario:
40                      if simbolos_produccion not in reglas_diccionario[cabeza]:
41                          reglas_diccionario[cabeza].append(simbolos_produccion)
42                  else:
43                      reglas_diccionario[cabeza] = [simbolos_produccion]
44
45                  # Crear un nodo para la cabeza si no existe
46                  if cabeza not in nodos:
47                      nodo_cabeza = Nodo(cabeza, f"N{contador_nodos}")
48                      nodos[cabeza] = nodo_cabeza
49                      dot.node(nodo_cabeza.identificador, cabeza)
50                      contador_nodos += 1
51                      if raiz is None:
52                          raiz = nodo_cabeza
53
54                  # Obtener el nodo cabeza actual
55                  nodo_cabeza = nodos[cabeza]
56
57                  # Crear nodos para cada smbolo en la produccin y conectarlos correctamente
58                  for simbolo in simbolos_produccion:
59                      identificador_nodo_simbolo = f"{simbolo}_{contador_nodos}"
60                      nodo_simbolo = Nodo(simbolo, identificador_nodo_simbolo)
61                      nodos[identificador_nodo_simbolo] = nodo_simbolo
62                      dot.node(nodo_simbolo.identificador, simbolo)
63                      contador_nodos += 1
64
65                      # Conectar el nodo cabeza con el nodo smbolo
66                      nodo_cabeza.agregar_hijo(nodo_simbolo)
67                      dot.edge(nodo_cabeza.identificador, nodo_simbolo.identificador)
68
69                      # Asegurarse de que los nodos hijos tambin puedan tener relaciones correctas
70                      nodos[simbolo] = nodo_simbolo
71
72          # Verificar y agregar nodo epsilon para nodos hoja sin hijos que estn en no_terminales.txt
73          agregar_epsilon_a_hojas_sin_hijos(nodos, 'no_terminales.txt', dot)
74
75          # Resaltar nodos hojas sin hijos con relleno amarillo
76          resaltar_hojas_sin_hijos(nodos, dot)
77
78          dot.render(nombre_archivo, format='png', cleanup=True)
79          print(f"rbol  sintctico  guardado en {nombre_archivo}.png")
80
81      def agregar_epsilon_a_hojas_sin_hijos(nodos, no_terminales_file, dot):
82          with open(no_terminales_file, 'r') as file:
83              no_terminales = {line.strip() for line in file}
```

```
84
85          for nodo in nodos.values():
86              if not nodo.hijos and nodo.etiqueta in no_terminales:
87                  # Crear un nodo hijo con el smbolo epsilon
88                  identificador_nodo_epsilon = f"epsilon_{nodo.identificador}"
89                  nodo_epsilon = Nodo("ε", identificador_nodo_epsilon)
90                  nodo.agregar_hijo(nodo_epsilon)
91                  dot.node(nodo_epsilon.identificador, "ε", style='filled', fillcolor='yellow')
92                  dot.edge(nodo.identificador, nodo_epsilon.identificador)
93
94      def resaltar_hojas_sin_hijos(nodos, dot):
95          for nodo in nodos.values():
96              if not nodo.hijos:
97                  # Resaltar el nodo hoja con relleno amarillo
98                  dot.node(nodo.identificador, nodo.etiqueta, style='filled', fillcolor='yellow')
99
100     def main():
101         nombre_archivo_rastreo = 'rastreo.csv'
102         nombre_archivo_arbol = 'arbol_sintactico'
103
104         rastreo = cargar_rastreo(nombre_archivo_rastreo)
105         generar_arbol_sintactico(rastreo, nombre_archivo_arbol)
106
107     if __name__ == "__main__":
108         main()
```

12. 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

```
1       import ply.lex as lex
2
3       # Lista de tokens
4       tokens = (
5           'ENTERO', 'FLOTANTE', 'BOOLEANO', 'CADENA',
6           'MAS', 'MENOS', 'POR', 'ENTRE', 'IGUAL', 'IGUAL_IGUAL', 'DISTINTO',
7           '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'\['
    t_LISTA_CERRAR = r'\]'
    t_DICCIONARIO_ABRIR = r'\{'
    t_DICCIONARIO_CERRAR = r'\}'
    t_COMA = r','
    t_PUNTO = r'\.'

    # Operadores lgicos
    t_Y = r'y'
    t_O = r'o'
    t_NO = r'no'

    # Definicin de las reglas de los tokens ms complejos
    def t_FLOTANTE(t):
```

```
        r'-?\d+\.\d+'
        t.value = float(t.value)
        return t

    def t_ENTERO(t):
        r'-?\d+'
        t.value = int(t.value)
        return t

    def t_BOOLEANO(t):
        r'True|False'
        t.value = True if t.value == 'True' else False
        return t

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

    def t_IDENTIFICADOR(t):
        r'[a-zA-Z_][a-zA-Z_0-9]*'
        t.type = reserved.get(t.value, 'IDENTIFICADOR')
        return t

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

    def t_newline(t):
        r'\n'
        t.lexer.lineno += 1

    t_ignore = ' \t'

    def t_error(t):
        print(f"Carcter ilegal: {t.value[0]}")
        t.lexer.skip(1)

    # Construir el lexer
    lexer = lex.lex()

    # Prueba del lexer con condicional
    data = '''
    def f(x,g(c,7),h(g,k,l())) {
    }
    '''

    lexer.input(data)

    # Tokenizar e imprimir solo los tipos de tokens separados por un espacio
    tokens_list = []

    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))
```

13. 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

```
1       #Asignacion de variable
2       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)

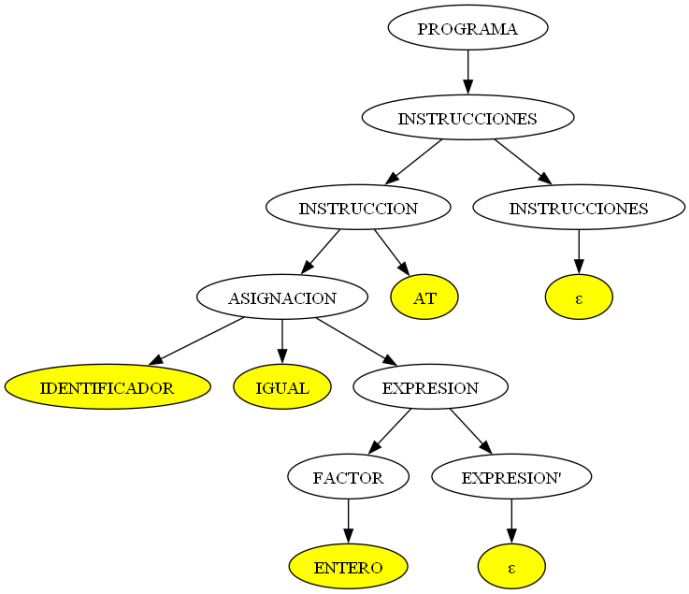

Figura 1: Árbol Sintáctico Generado

Listing 7: Ejmplo 02

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

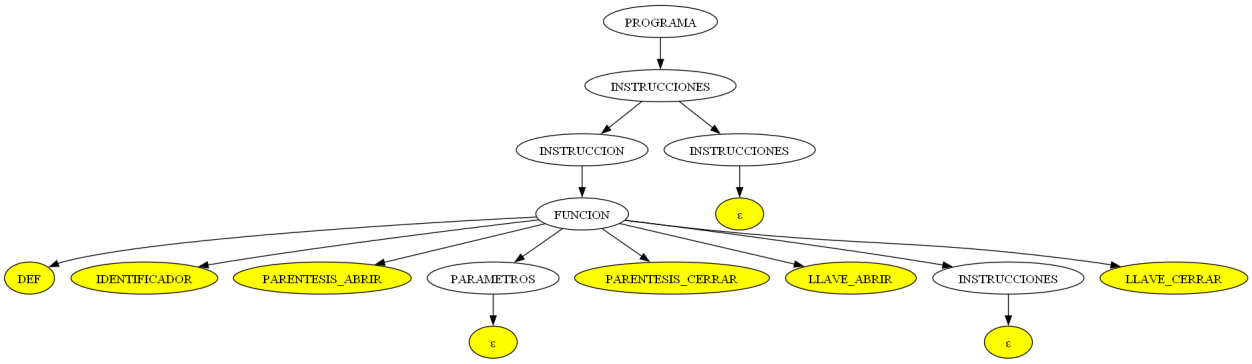

Figura 2: Árbol Sintáctico Generado

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

14. Tabla de símbolos

Este código es un parser que genera un árbol sintáctico a partir de una gramática y una secuencia de tokens, y maneja una tabla de símbolos para almacenar las variables y funciones encontradas durante el análisis. A continuación, se describen los componentes clave del código:

- **Generación del Árbol Sintáctico:** La función `obtener_raiz` crea el árbol sintáctico a partir de la gramática y los tokens proporcionados. Utiliza una función recursiva `parser` que intenta emparejar las producciones de la gramática con los tokens disponibles. Cada emparejamiento exitoso crea un nodo en el árbol sintáctico.
- **Clase Nodo:** Los nodos del árbol sintáctico son representados por la clase `Nodo`, que almacena etiquetas, valores y una lista de hijos. Esta clase facilita la construcción del árbol mediante la agregación de nodos hijos.
- **Tabla de Símbolos:** La `TablaDeSimbolos` almacena los símbolos encontrados durante el análisis, como variables y funciones. Cada símbolo tiene un nombre, tipo, valor y ámbito. Se asegura de que no haya duplicados dentro de un mismo ámbito.
- **Agregación de Símbolos:** La función `agregarSimbolos` recorre el árbol sintáctico, buscando identificadores y funciones, y agrega los símbolos encontrados a la tabla de símbolos. También maneja los parámetros de las funciones y las actualizaciones de las variables al procesar asignaciones.
- **Manejo de Asignaciones:** En el caso de las asignaciones, el código busca el valor a asignar y actualiza el valor de la variable en la tabla de símbolos.
- **Funciones de Impresión:** Existen funciones como `imprimir_tabla` para mostrar la tabla de símbolos, y `imprimirNodos` para imprimir los valores de los nodos del árbol sintáctico.

Listing 8: Tabla$_S$*imbolos.py*

```
import re
from Generar_Arbol import Nodo, cargar_gramatica, cargar_tokens, cargar_no_terminales

def obtener_raiz(gramatica, tokens, no_terminales):
    contador_nodos = 0

    def agregar_nodo(etiqueta, valor=None):
        nonlocal contador_nodos
        nodo = Nodo(etiqueta, f'N{contador_nodos}', valor)
        contador_nodos += 1
        return nodo

    def parser(no_terminal, tokens, index):
        if no_terminal not in gramatica:
            print(f"No se encuentra el no terminal: {no_terminal}")
            return None, index

        producciones = sorted(gramatica[no_terminal], key=lambda p: evaluar_prioridad(p, tokens,
            index), reverse=True)
        for produccion in producciones:
            hijos = []
            nuevo_index = index
            exito = True
            for simbolo in produccion:
                if simbolo in gramatica:
                    nodo_hijo, nuevo_index = parser(simbolo, tokens, nuevo_index)
                    if nodo_hijo is None:
                        exito = False
                        break
                    hijos.append(nodo_hijo)
                else:
                    if simbolo == "''":
                        continue
                    if nuevo_index < len(tokens) and re.match(f'{simbolo}:.*',
                        tokens[nuevo_index]):
```

```
                            token_tipo, token_valor = tokens[nuevo_index].split(':')
                            nodo_hijo = agregar_nodo(token_tipo, token_valor)
                            hijos.append(nodo_hijo)
                            nuevo_index += 1
                        else:
                            exito = False
                            break

                if exito:
                    nodo_actual = agregar_nodo(no_terminal)
                    for hijo in hijos:
                        nodo_actual.agregar_hijo(hijo)
                    return nodo_actual, nuevo_index

            if "'" in [simbolo for produccion in gramatica[no_terminal] for simbolo in produccion]:
                return agregar_nodo(no_terminal), index

            return None, index

        def evaluar_prioridad(produccion, tokens, index):
            coincidencias = 0
            for simbolo in produccion:
                if index + coincidencias < len(tokens):
                    token = tokens[index + coincidencias]
                    if simbolo in gramatica or re.match(f'{simbolo}:.*', token):
                        coincidencias += 1
                    else:
                        break
                else:
                    break
            return coincidencias

        raiz, _ = parser('PROGRAMA', tokens, 0)
        return raiz


    class Simbolo:
        def __init__(self, nombre, tipo, es_funcion=False, parametros=None, valor=None,
             ambito="Global"):
            self.nombre = nombre
            self.tipo = tipo
            self.es_funcion = es_funcion
            self.parametros = parametros
            self.valor = valor
            self.ambito = ambito

        def __repr__(self):
            if self.es_funcion:
                return f"Funcin(nombre={self.nombre}, tipo={self.tipo}, parmetros={self.parametros},
                     valor={self.valor})"
            else:
                return f"Variable(nombre={self.nombre}, tipo={self.tipo}, valor={self.valor})"


    class TablaDeSimbolos:
        def __init__(self):
            self.tabla = {}
            self.ambitos = []

        def agregar_simbolo(self, nombre, tipo, es_funcion=False, parametros=None, valor=None,
             ambito="Global"):
            if nombre in self.tabla:
                simbolo_existente = self.tabla[nombre]
                if simbolo_existente.ambito == ambito:
                    raise ValueError(f"El smbolo '{nombre}' ya est declarado en el mbito '{ambito}'.")
```

```
96
97              simbolo = Simbolo(nombre, tipo, es_funcion, parametros, valor, ambito)
98              self.tabla[nombre] = simbolo
99
100         def buscar_simbolo(self, nombre):
101             return self.tabla.get(nombre, None)
102
103         def declarar_ambito(self):
104             self.ambitos.append(set())
105
106         def finalizar_ambito(self):
107             if self.ambitos:
108                 for simbolo in self.ambitos.pop():
109                     del self.tabla[simbolo]
110             else:
111                 raise ValueError("No hay mbitos abiertos para finalizar.")
112
113         def agregar_variable_local(self, nombre, tipo, valor=None):
114             if not self.ambitos:
115                 raise ValueError("No hay un mbito activo para declarar la variable.")
116
117             if nombre in self.tabla:
118                 raise ValueError(f"El smbolo '{nombre}' ya est declarado.")
119
120             simbolo = Simbolo(nombre, tipo, es_funcion=False, valor=valor)
121             self.tabla[nombre] = simbolo
122             self.ambitos[-1].add(nombre)
123
124         def imprimir_tabla(self):
125             print("\n" + "="*50)
126             print(f"{'Smbolos de la Tabla de Smbolos':^50}")
127             print("="*50)
128
129             for simbolo in self.tabla.values():
130                 print(f"\n{'Nombre:':<15} {simbolo.nombre}")
131                 print(f"{'Tipo:':<15} {simbolo.tipo}")
132                 print(f"{'Es funcin:':<15} {simbolo.es_funcion}")
133                 print(f"{'Parmetros:':<15} {simbolo.parametros if simbolo.parametros else 'N/A'}")
134                 print(f"{'Valor:':<15} {simbolo.valor if simbolo.valor is not None else 'N/A'}")
135                 print(f"{'mbito:':<15} {simbolo.ambito}")
136                 print("-"*50)
137
138             print("="*50)
139
140     TablaSimbolos = TablaDeSimbolos()
141
142     def buscarNodo(nodo, nombre):
143         if nodo is None:
144             return None
145         if nodo.etiqueta == nombre:
146             return nodo
147         for hijo in nodo.hijos:
148             encontrado = buscarNodo(hijo, nombre)
149             if encontrado:
150                 return encontrado
151         return None
152
153     def buscarValor(nodo):
154         if nodo.etiqueta == "RETORNAR":
155             return obtener_hermanos(nodo)
156         for hijo in nodo.hijos:
157             resultado = buscarValor(hijo)
158             if resultado:
159                 return resultado
160         return None
```

```
161
162         def obtener_hermanos(nodo):
163             hermanos = []
164             if nodo.padre:
165                 index = nodo.padre.hijos.index(nodo)
166                 hermanos = nodo.padre.hijos[index + 1:]
167             valores_hijos = []
168             for hermano in hermanos:
169                 valores_hijos.extend(obtener_descendientes(hermano))
170             return ''.join([str(valor) for valor in valores_hijos if valor is not None and valor != '@'])
171
172         def obtener_descendientes(nodo):
173             valores = []
174             if nodo.valor is not None and nodo.valor != '@':
175                 valores.append(nodo.valor)
176             for hijo in nodo.hijos:
177                 valores.extend(obtener_descendientes(hijo))
178             return valores
179
180         def agregarSimbolos(nodo, ambito="Global"):
181             if nodo.etiqueta == "FUNCION":
182                 valor = buscarValor(nodo)
183                 parametros = BuscarParametros(buscarNodo(nodo, "PARAMETROS"))
184                 nombre = buscarNodo(nodo, "IDENTIFICADOR")
185                 if TablaSimbolos.buscar_simbolo(nombre.valor) is None:
186                     TablaSimbolos.agregar_simbolo(nombre.valor, "Function", True, parametros, valor,
                            ambito)
187                 ambito_funcion = nombre.valor
188                 for parametro in parametros:
189                     if TablaSimbolos.buscar_simbolo(parametro) is None:
190                         TablaSimbolos.agregar_simbolo(parametro, "Variable", False, None, None,
                                ambito_funcion)
191                 instrucciones = buscarNodo(nodo, "INSTRUCCIONES")
192                 if instrucciones:
193                     for hijo in instrucciones.hijos:
194                         agregarSimbolos(hijo, ambito_funcion)
195             elif nodo.etiqueta == "ASIGNACION":
196                 identificador = buscarNodo(nodo, "IDENTIFICADOR")
197                 if identificador:
198                     valor = buscar_valores_asignacion(nodo)
199                     if TablaSimbolos.buscar_simbolo(identificador.valor) is None:
200                         TablaSimbolos.agregar_simbolo(identificador.valor, "Variable", False, None,
                                valor, ambito)
201                     else:
202                         simbolo = TablaSimbolos.buscar_simbolo(identificador.valor)
203                         simbolo.valor = valor
204             for hijo in nodo.hijos:
205                 agregarSimbolos(hijo, ambito)
206
207         def BuscarParametros(nodo):
208             parametros = []
209             if nodo.etiqueta == "IDENTIFICADOR":
210                 parametros.append(nodo.valor)
211             for hijo in nodo.hijos:
212                 parametros.extend(BuscarParametros(hijo))
213             return parametros
214
215         def buscar_valores_asignacion(nodo):
216             valores = []
217             if nodo.etiqueta == "ASIGNACION":
218                 igual = buscarNodo(nodo, "IGUAL")
219                 if igual and igual.padre:
220                     padre = igual.padre
221                     index_igual = padre.hijos.index(igual)
222                     hermanos_despues_igual = padre.hijos[index_igual + 1:]
```

```
                for hermano in hermanos_despues_igual:
                    valores.extend(obtener_descendientes(hermano))
        return ''.join([str(valor) for valor in valores if valor is not None and valor != '@'])

    def imprimirNodos(nodo):
        for hijo in nodo.hijos:
            if hijo.valor:
                print(f"Valor: {hijo.valor}")
            imprimirNodos(hijo)

    def main():
        gramatica = cargar_gramatica("gramatica.txt")
        tokens = cargar_tokens("tokens.txt")
        no_terminales = cargar_no_terminales("no_terminales.txt")

        raiz = obtener_raiz(gramatica, tokens, no_terminales)

        agregarSimbolos(raiz)
        TablaSimbolos.imprimir_tabla()

        #imprimirNodos(raiz)

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

15. Código SPIM

Este código define una clase llamada `SPIMGenerator`, la cual se encarga de generar código en lenguaje ensamblador SPIM a partir de un programa escrito en Python. Para ello, se utiliza el módulo `ast` de Python para analizar el código fuente y recorrer sus componentes, como asignaciones, expresiones aritméticas, condicionales `if` y llamadas a la función `print`.

15.1. Funcionamiento

- **Asignaciones de variables**: Cuando se encuentra una asignación en el código Python, el generador asigna un espacio de memoria en la sección de datos y genera una instrucción para almacenar el valor de la variable en esa posición de memoria.
- **Operaciones aritméticas**: Se manejan operaciones como suma, resta, multiplicación y división. Dependiendo del tipo de operación, se generan las instrucciones adecuadas en MIPS para realizar los cálculos.
- **Condicionales `if`**: El generador convierte las estructuras condicionales en bloques de código con saltos (`beq`, `bne`, etc.) que permiten decidir qué parte del código ejecutar dependiendo de la condición.
- **Impresión de valores**: La función `print` se traduce en llamadas al sistema MIPS para imprimir valores. Si el valor es un número o una cadena, el generador lo convierte a la instrucción adecuada en MIPS para su impresión en consola.

15.2. Estructura del Código

- **Sección de datos**: Contiene las variables y las cadenas que se usan en el programa.
- **Sección de texto**: Contiene las instrucciones MIPS generadas a partir de las expresiones y estructuras de control del código Python.

Finalmente, el código generado es escrito en un archivo de texto con extensión `.asm`, que puede ser utilizado en un simulador SPIM para su ejecución.

Listing 9: CodigoSPIM.py

```
import ast

class SPIMGenerator(ast.NodeVisitor):
    def __init__(self):
        self.variables = {}
        self.label_count = 0
        self.data_section = []
        self.text_section = []
        self.string_count = 0 # Contador para cadenas
        self.strings = {} # Mapa de cadenas a etiquetas
        self.registers = [f"$t{i}" for i in range(10)] # $t0 a $t9
        self.register_pool = self.registers.copy()

    def new_label(self, base):
        label = f"{base}_{self.label_count}"
        self.label_count += 1
        return label

    def new_string_label(self):
        label = f"str_{self.string_count}"
        self.string_count += 1
        return label

    def allocate_register(self):
        if not self.register_pool:
            raise RuntimeError("No hay registros disponibles.")
        return self.register_pool.pop(0)

    def free_register(self, reg):
        if reg in self.registers and reg not in self.register_pool:
            self.register_pool.insert(0, reg)

    def generate(self, code):
        tree = ast.parse(code)
        self.visit(tree)
        # Combine data and text sections
        data = "\n".join(self.data_section + list(self.strings.values()))
        text = "\n".join(self.text_section)
        full_code = f".data\n{data}\n\n.text\n.globl main\nmain:\n{text}\n li $v0, 10\n syscall"
        return full_code

    def visit_Module(self, node):
        for stmt in node.body:
            self.visit(stmt)

    def visit_Assign(self, node):
        # Asumimos una sola variable por asignacin
        var_name = node.targets[0].id
        var_label = f"var_{var_name}" # Prefijo para evitar colisiones
        if var_name not in self.variables:
            self.variables[var_name] = var_label
            self.data_section.append(f"{var_label}: .word 0")
        # Evaluar la expresin y obtener el registro que contiene el resultado
        result_reg = self.visit(node.value)
        # Almacenar el resultado en la variable
        self.text_section.append(f" sw {result_reg}, {var_label}")
        # Liberar el registro temporal
        self.free_register(result_reg)

    def visit_Expr(self, node):
```

```
61                 self.visit(node.value)
62
63             def visit_BinOp(self, node):
64                 # Evaluar el operando izquierdo
65                 left_reg = self.visit(node.left)
66                 # Evaluar el operando derecho
67                 right_reg = self.visit(node.right)
68                 # Asignar un registro para el resultado
69                 result_reg = self.allocate_register()
70                 # Generar la instruccin correspondiente
71                 if isinstance(node.op, ast.Add):
72                     self.text_section.append(f" add {result_reg}, {left_reg}, {right_reg}")
73                 elif isinstance(node.op, ast.Sub):
74                     self.text_section.append(f" sub {result_reg}, {left_reg}, {right_reg}")
75                 elif isinstance(node.op, ast.Mult):
76                     self.text_section.append(f" mul {result_reg}, {left_reg}, {right_reg}")
77                 elif isinstance(node.op, ast.Div):
78                     self.text_section.append(f" div {left_reg}, {right_reg}")
79                     self.text_section.append(f" mflo {result_reg}")
80                 else:
81                     raise NotImplementedError(f"Operacin {type(node.op)} no soportada")
82                 # Liberar los registros de los operandos
83                 self.free_register(left_reg)
84                 self.free_register(right_reg)
85                 return result_reg
86
87             def visit_Num(self, node):
88                 reg = self.allocate_register()
89                 self.text_section.append(f" li {reg}, {node.n}")
90                 return reg
91
92             def visit_Name(self, node):
93                 var_label = f"var_{node.id}" # Prefijo para evitar colisiones
94                 if node.id not in self.variables:
95                     self.variables[node.id] = var_label
96                     self.data_section.append(f"{var_label}: .word 0")
97                 reg = self.allocate_register()
98                 self.text_section.append(f" lw {reg}, {var_label}")
99                 return reg
100
101            def visit_If(self, node):
102                # Generar etiquetas nicas para este bloque if-else
103                label_if_true = self.new_label("if_true")
104                label_if_false = self.new_label("if_false")
105                label_if_end = self.new_label("if_end")
106
107                # Evaluar la condicin
108                if isinstance(node.test, ast.Compare):
109                    left_reg = self.visit(node.test.left)
110                    right_reg = self.visit(node.test.comparators[0])
111                    op = node.test.ops[0]
112                    # Generar la comparacin y salto condicional
113                    if isinstance(op, ast.Eq):
114                        self.text_section.append(f" beq {left_reg}, {right_reg}, {label_if_true}")
115                    elif isinstance(op, ast.NotEq):
116                        self.text_section.append(f" bne {left_reg}, {right_reg}, {label_if_true}")
117                    elif isinstance(op, ast.Lt):
118                        self.text_section.append(f" blt {left_reg}, {right_reg}, {label_if_true}")
119                    elif isinstance(op, ast.LtE):
120                        self.text_section.append(f" ble {left_reg}, {right_reg}, {label_if_true}")
121                    elif isinstance(op, ast.Gt):
122                        self.text_section.append(f" bgt {left_reg}, {right_reg}, {label_if_true}")
123                    elif isinstance(op, ast.GtE):
124                        self.text_section.append(f" bge {left_reg}, {right_reg}, {label_if_true}")
125                    else:
```

```
126                     raise NotImplementedError(f"Operador de comparacin {type(op)} no soportado")
127                 self.text_section.append(f" j {label_if_false}")
128                 # Bloque if
129                 self.text_section.append(f"{label_if_true}:")
130                 for stmt in node.body:
131                     self.visit(stmt)
132                 self.text_section.append(f" j {label_if_end}")
133                 # Bloque else
134                 self.text_section.append(f"{label_if_false}:")
135                 for stmt in node.orelse:
136                     self.visit(stmt)
137                 # Fin del if
138                 self.text_section.append(f"{label_if_end}:")
139                 # Liberar los registros de la condicin
140                 self.free_register(left_reg)
141                 self.free_register(right_reg)
142             else:
143                 raise NotImplementedError("Solo se soportan comparaciones en condiciones if")
144 
145         def visit_Call(self, node):
146             if isinstance(node.func, ast.Name) and node.func.id == 'print':
147                 if len(node.args) != 1:
148                     raise NotImplementedError("Solo se soporta print con un argumento")
149                 arg = node.args[0]
150                 if isinstance(arg, ast.Str):
151                     # Imprimir cadena
152                     label = self.get_string_label(arg.s)
153                     self.text_section.append(f" la $a0, {label}")
154                     self.text_section.append(" li $v0, 4")
155                     self.text_section.append(" syscall")
156                     # Imprimir nueva lnea
157                     self.text_section.append(" li $a0, 10")
158                     self.text_section.append(" li $v0, 11")
159                     self.text_section.append(" syscall")
160                 elif isinstance(arg, ast.Num):
161                     # Imprimir entero
162                     reg = self.visit(arg)
163                     self.text_section.append(f" move $a0, {reg}")
164                     self.text_section.append(" li $v0, 1")
165                     self.text_section.append(" syscall")
166                     # Imprimir nueva lnea
167                     self.text_section.append(" li $a0, 10")
168                     self.text_section.append(" li $v0, 11")
169                     self.text_section.append(" syscall")
170                     self.free_register(reg)
171                 elif isinstance(arg, ast.Name):
172                     # Imprimir variable
173                     reg = self.visit(arg)
174                     self.text_section.append(f" move $a0, {reg}")
175                     self.text_section.append(" li $v0, 1")
176                     self.text_section.append(" syscall")
177                     # Imprimir nueva lnea
178                     self.text_section.append(" li $a0, 10")
179                     self.text_section.append(" li $v0, 11")
180                     self.text_section.append(" syscall")
181                     self.free_register(reg)
182                 else:
183                     raise NotImplementedError("Tipo de argumento en print no soportado")
184             else:
185                 raise NotImplementedError("Solo se soportan llamadas a la funcin print")
186 
187         def get_string_label(self, s):
188             if s in self.strings:
189                 return self.strings[s]
190             label = self.new_string_label()
```

```
            self.strings[s] = f"{label}: .asciiz \"{s}\""
            return label

    # Ejemplo de uso con if-else anidados
    if __name__ == "__main__":
        python_code = """
    a = 20 + 2
    b = 30
    c = a + b
    print(c)
    """
        generator = SPIMGenerator()
        spim_code = generator.generate(python_code)
        with open("output.asm", "w") as f:
            f.write(spim_code)
        print("Cdigo SPIM generado en 'output.asm'")
```

15.3. Ejemplo

Con esta entrada:

Listing 10: Ejemplo 01

```
a = 20 + 2
b = 30
c = a + b
print(c)
```

Genera:

Listing 11: Ejemplo 01

```
.data
var_a: .word 0
var_b: .word 0
var_c: .word 0

.text
.globl main
main:
    li $t0, 20
    li $t1, 2
    add $t2, $t0, $t1
    sw $t2, var_a
    li $t2, 30
    sw $t2, var_b
    lw $t2, var_a
    lw $t1, var_b
    add $t0, $t2, $t1
    sw $t0, var_c
    lw $t0, var_c
    move $a0, $t0
    li $v0, 1
    syscall
    li $a0, 10
    li $v0, 11
    syscall
    li $v0, 10
    syscall
```