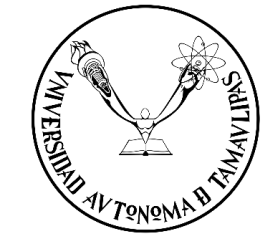
Universidad Autónoma de Tamaulipas
Facultad de Ingeniería Tampico



ASIGNATURA
Programación de Sistemas de Base II

9no. Semestre – Grupo “J”
2025 - 3

Kobra V2
Manual Técnico y de Usuario

Docente: Dante Adolfo Muñoz Quintero

Integrantes:
Badillo González Víctor Manuel
Clemente Villegas José Adán
Cristóbal Francisco Jesús Marcelino
Granados Gallegos Carlo Sebastián

Índice

1. Introducción 4

Visión general 4

Propósito y justificación 4

2. Instalación y configuración 5

Requisitos del sistema 5

Dependencias necesarias 6

Descarga y estructura del proyecto 6

Instalación paso a paso 6

3. Guía de uso 7

Uso básico de la Línea de Comandos 7

Workflow de transpilación 8

Formato del archivo de entrada 8

Formato de archivo de salida 9

Manejo de errores 9

Ejemplos de uso 11

Ejemplo: Declaración de variables y operadores 11
Ejemplo: Condicionales 11
Ejemplo: Ciclo MIENTRAS 11
Ejemplo: Ciclo PARA 11
Ejemplo Manejo de cadenas y concatenación 12
Ejemplo práctico que integra multiples funciones del lenguaje: 12

Muestra de funcionamiento 13

Manejo de errores 31

4. Referencia técnica 37

Resumen del lenguaje 37

Características clave del lenguaje 37

Arquitectura del lenguaje 37

Estructura del programa 38

Tipos de sentencias 38
Estructura de bloques 39

Descripción General de las Características del Lenguaje 39
Sistema de Tipos 39
Control de flujo 40
Expresiones y operadores 40

Elementos Léxicos 41
Palabras clave 41
Literales 41
Identificadores 41

Convenciones de Sintaxis 42
Terminación de Sentencias 42
Delimitadores de Bloque 42
Paréntesis en Expresiones 42

5. Código fuente 43

KobraV2.g4 43

KobraCompiler.py 45

main.py 50

6. Troubleshooting 51

7. Preguntas frecuentes 52

1. Introducción

Visión general

KobraV2 es un sistema de transpilación fuente-a-fuente, lo que significa que su función principal no es generar código máquina ni bytecode, sino transformar un lenguaje de programación de alto nivel a otro lenguaje de alto nivel, en este caso Python. Esta característica permite que los usuarios escriban programas usando palabras clave en español, y que estos programas puedan ejecutarse inmediatamente en Python, aprovechando su entorno de ejecución, librerías y manejo de memoria.

El sistema procesa archivos de texto plano que cumplen con la gramática definida para KobraV2. Cada archivo fuente es sometido a un proceso de análisis léxico y sintáctico, donde se identifican los tokens y se construye un árbol de análisis sintáctico que refleja la estructura jerárquica del programa. Posteriormente, un módulo de generación de código recorre este árbol y produce un archivo `.py` que mantiene la lógica original del programa, adaptando construcciones de KobraV2 a Python de manera coherente y segura.

Propósito y justificación

El desarrollo de KobraV2 surge de la necesidad de mitigar la barrera idiomática que se llega a enfrentar los hispanohablantes al aprender programación. Los lenguajes tradicionales, como Python, Java o C++, utilizan palabras clave en inglés, lo que obliga a los principiantes a aprender simultáneamente la lógica de programación y la sintaxis de un idioma extranjero. KobraV2 busca simplificar este proceso al ofrecer un lenguaje donde las palabras clave y las instrucciones están completamente en español, permitiendo que los estudiantes se concentren primero en comprender los conceptos fundamentales de la programación.

Accesibilidad

KobraV2 permite aprender conceptos lógicos y estructurales utilizando palabras clave nativas como *ENTERO, FLOTANTE, SI, SINO, MIENTRAS, PARA* e *IMPRIMIR*. Al usarse en el idioma español se puede internalizar más fácilmente la semántica de variables, condicionales y bucles, sin verse abrumados por la traducción mental que implica un lenguaje en inglés. Esta accesibilidad promueve una experiencia de aprendizaje más fluida, intuitiva y menos intimidante.

Abstracción y disciplina de tipos

Aunque KobraV2 simplifica ciertos aspectos de la programación, manteniendo la disciplina de tipos. La declaración de variables como ENTERO, FLOTANTE, CADENA o BOOLEANO, ayuda a entender la diferencia entre tipos de datos y la importancia de la tipificación en la programación.

Esta característica es pedagógicamente valiosa, ya que establece una base sólida antes de que los alumnos migren a lenguajes de tipado dinámico como Python o JavaScript, donde las reglas de tipos son más flexibles.

Motivación pedagógica

El propósito de KobraV2 es crear un puente entre la enseñanza inicial de la programación y la práctica profesional. Los estudiantes aprenden a estructurar programas, controlar el flujo de ejecución y manipular datos usando un lenguaje que les resulta familiar, pero al mismo tiempo producen código que se ejecuta en un entorno real de Python. Esto permite una transición natural hacia lenguajes más complejos, proporcionando confianza y comprensión profunda de los conceptos fundamentales.

2. Instalación y configuración

Esta sección describe los requisitos, pasos de instalación y configuraciones necesarias para utilizar correctamente el transpilador KobraV2.

Requisitos del sistema

Para garantizar el correcto funcionamiento del compilador, se recomienda contar con los siguientes elementos:

- Python 3.x instalado (versión recomendada: 3.10 o superior).
- Sistema operativo compatible:
  - Windows 10/11
  - Linux (cualquier distribución con soporte para Python)
  - macOS
- Entorno con acceso a línea de comandos (CMD, PowerShell, Terminal).
- Biblioteca runtime de ANTLR para Python, necesaria para ejecutar el lexer y parser generados.
- Opcionalmente, el usuario puede instalar el JAR de ANTLR si desea modificar la gramática y regenerar los analizadores.

Dependencias necesarias

KobraV2 requiere la instalación de los siguientes componentes:

Componente	Versión	Propósito
Python	3.x	Entorno de ejecución
ANTLR4 Runtime	4.13.2	Biblioteca de tiempo de ejecución del analizador
Archivos de gramática KobraV2	Generados	Componentes del analizador léxico y sintáctico

Descarga y estructura del proyecto

Para obtener el sistema, el usuario debe clonar o descargar el repositorio:

```
git clone https://github.com/badillovictor/KobraV2.git
cd KobraV2
```

La estructura del proyecto incluye:

- KobraV2.g4 – archivo de gramática del lenguaje.
- KobraV2Lexer.py – archivo generado por ANTLR que define los tokens.
- KobraV2Parser.py – parser generado a partir de la gramática.
- KobraCompiler.py – módulo principal donde se implementa el visitor y generación de código.
- main.py – archivo que ejecuta el proceso de transpilación.

Esta organización permite identificar claramente qué partes del sistema pueden modificarse y cuáles son automáticas.

Instalación paso a paso

1. Instalar Python 3.x

Descargar desde la página oficial: https://www.python.org/downloads/

Verificar la instalación con `python -version` o `python3 --version`

2. Instalar ANTLR4 Python Runtime

```
pip install antlr4-python3-runtime==4.13.2
```

3. Verificar archivos requeridos

Comprueba que los siguientes archivos están en tu directorio de trabajo:

- `main.py` – Script de punto de entrada.
- `KobraCompiler.py` – Implementación del transpilador.
- `KobraV2Lexer.py` – Analizador léxico generado.
- `KobraV2Parser.py` – Analizador sintáctico generado.
- `KobraV2Visitor.py` – Clase base de visitante generada.

3. Guía de uso

Esta sección explica cómo utilizar el transpilador KobraV2, mostrando el flujo completo: escribir un programa en el lenguaje Kobra, transpilado a Python y ejecución del archivo resultante.

Uso básico de la Línea de Comandos

El transpilador se ejecuta a través de main.py con un único argumento de línea de comandos que especifica la ruta del archivo de entrada:

```
python main.py <archivo_de_entrada.txt>
```

Argumento	Tipo	Requerido	Descripción
<archivo_de_entrada.txt>	String	Si	Ruta del archivo fuente de KobraV2

El archivo de entrada debe tener la extensión *.txt*. El transpilador genera automáticamente un archivo de salida con el mismo nombre base, pero con la extensión *.py* (lenguaje Python).

Ejemplo de ejecución

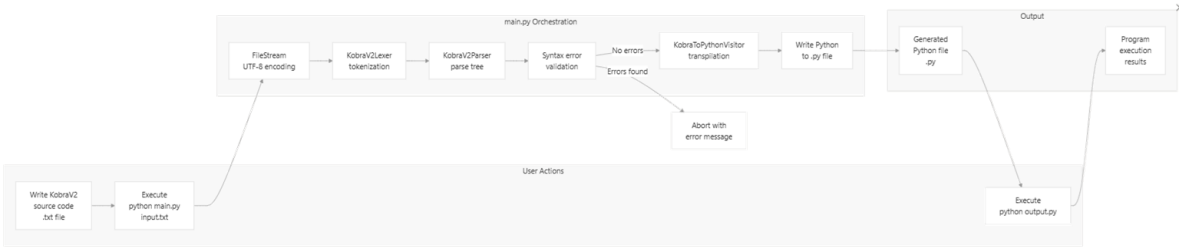
```
python main.py saludo.txt
```

Este commando:

- Lee `saludo.txt` como entrada.
- Analiza y valida el código fuente de KobraV2.
- Transpila a Python.
- Escribe la salida en `saludo.py`

Workflow de transpilación

Workflow de transpilación de extremo a extremo desde el código fuente hasta la ejecución.



Formato del archivo de entrada

Especificaciones del archivo

Propiedad	Requerimiento
Extensión del archivo	`.txt`
Codificación de caracteres	UTF-8
Sintaxis	Especificación del lenguaje KobraV2
Comentarios	Soportados mediante `//`

Estructura del archivo de entrada

Los archivos fuente de KobraV2 siguen esta estructura general:

```
// Comentarios (opcional)
TIPO variable = valor;
…
instrucciones;
…
```

Ejemplo de archivo de entrada

El siguiente ejemplo muestra una estructura válida de archivo de entrada para KobraV2:

```
// Calculadora de cuenta de restaurante
```

```
FLOTANTE costoComida = 450.50;
FLOTANTE costoBebidas = 120.00;
FLOTANTE total = 0.0;

total = costoComida + costoBebidas;
IMPRIMIR(total);
```

Formato de archivo de salida

Propiedad	Requerimiento
Extensión del archivo	`.py`
Codificación de caracteres	UTF-8
Nombre base	Igual al archivo de entrada
Ubicación	Mismo directorio que el archivo de entrada

Estructura del archivo de salida

Los archivos de Python generados contienen:

4. Función auxiliar: La función `_kobra_div()` para manejar la semántica de la división.
5. Sentencias traducidas: Las sentencias de KobraV2 convertidas a código Python.
6. Indentación correcta: Indentación compatible con Python para las estructuras de control.

Manejo de errores

El transpilador valida la sintaxis de KobraV2 antes de la generación de código. Si se detectan errores de sintaxis, el transpilador detiene la ejecución y muestra un mensaje de error:

```
Errores de sintaxis encontrados. No se generó código
```

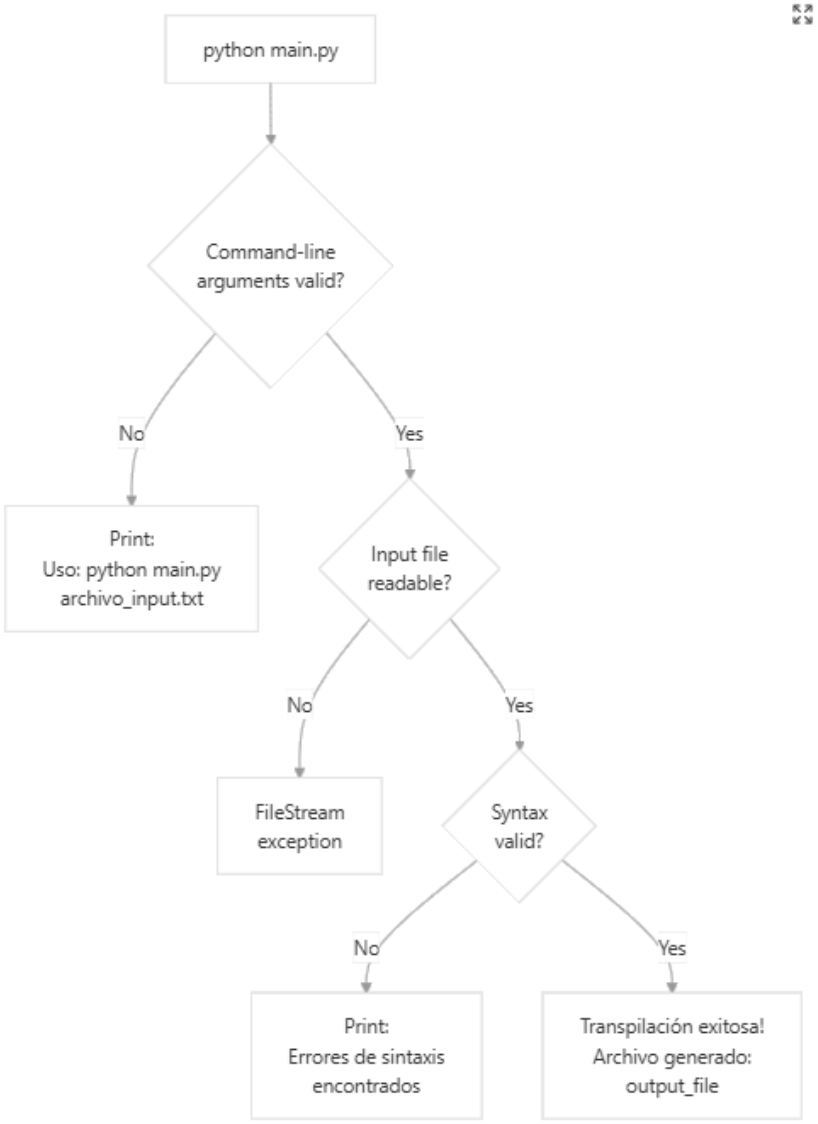
La validación ocurre después del análisis sintáctico:

```
if parser.getNumberOfSyntaxErrors() > 0:
    print("Errores de sintaxis encontrados. No se generó código.")
    return
```

No se crea ningún archivo de salida cuando hay errores de sintaxis presentes.

Flujo de manejo de errores

Puntos de detección y manejo de errores en el flujo de transpilación



Ejemplos de uso

Ejemplos de dificultad creciente para mostrar el funcionamiento del lenguaje.

Ejemplo: Declaración de variables y operadores

```
ENTERO a = 10;
ENTERO b = 3;
ENTERO c = a + b;
IMPRIMIR(c);
```

Ejemplo: Condicionales

```
ENTERO x = 5;
SI (x > 0) {
    IMPRIMIR("El valor es positivo");
} SINO {
    IMPRIMIR("El valor es negativo");
}
```

Ejemplo: Ciclo MIENTRAS

```
ENTERO i = 1;
MIENTRAS (i <= 5) {
    IMPRIMIR(i);
    i = i + 1;
}
```

Ejemplo: Ciclo PARA

```
PARA (i = 0; i < 3; i = i + 1) {
    IMPRIMIR("Iteración " + i);
}
```

Ejemplo Manejo de cadenas y concatenación

```
CADENA nombre = "Kobra";
IMPRIMIR("Hola " + nombre);
```

Ejemplo práctico que integra multiples funciones del lenguaje:

```
// --- CALCULADORA DE CUENTA RESTAURANTE ---
FLOTANTE costoComida = 450.50;
FLOTANTE costoBebidas = 120.00;
FLOTANTE tasaPropina = 0.15;
ENTERO personas = 3;
FLOTANTE total = 0.0;
FLOTANTE propina = 0.0;
FLOTANTE totalFinal = 0.0;
FLOTANTE porPersona = 0.0;

total = costoComida + costoBebidas;

IMPRIMIR("--- TICKET DE KOBRA RESTAURANT ---");
IMPRIMIR("Subtotal comida y bebida:");
IMPRIMIR(total);

// Logica de propina alta si el consumo fue alto
SI (total > 500.0) {
    IMPRIMIR("Consumo alto detectado. Aplicando propina sugerida
del 15%");
    propina = total * tasaPropina;
} SINO {
```

```
        IMPRIMIR("Consumo estandar. Propina fija de 50 pesos.");
        propina = 50.0;
    }
    totalFinal = total + propina;
    IMPRIMIR("Propina calculada:");
    IMPRIMIR(propina);
    IMPRIMIR("TOTAL A PAGAR:");
    IMPRIMIR(totalFinal);
    porPersona = totalFinal / personas;
    IMPRIMIR("Monto a pagar por persona:");
    IMPRIMIR(porPersona);
    IMPRIMIR("Imprimiendo recibos individuales...");
    ENTERO i = 0;
    PARA (i = 0; i < personas; i = i + 1) {
        IMPRIMIR("Recibo impreso.");
    }
```

Muestra de funcionamiento

Ejemplo 1:

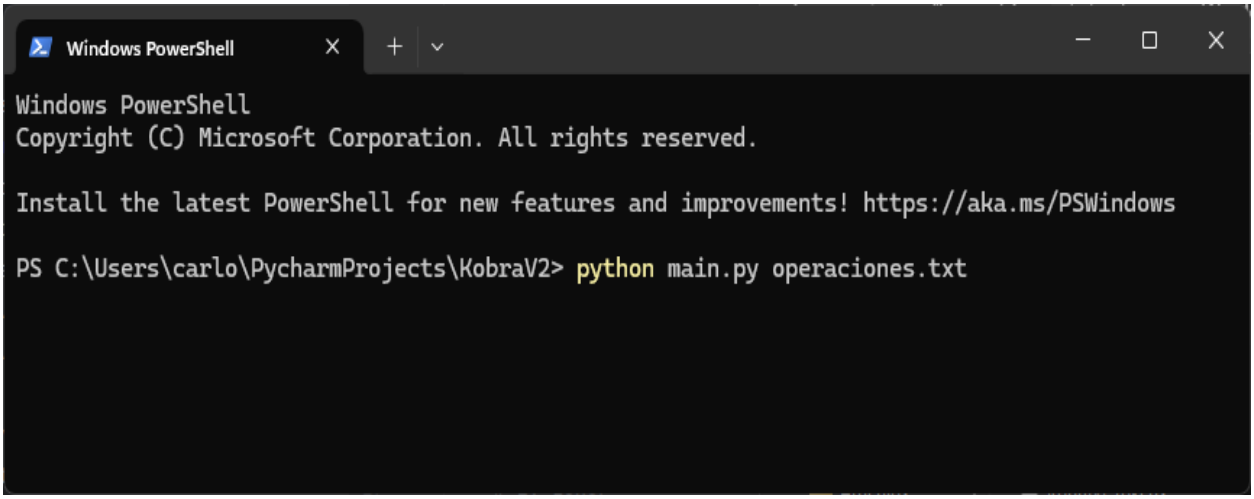
Entrada:

operaciones.txt

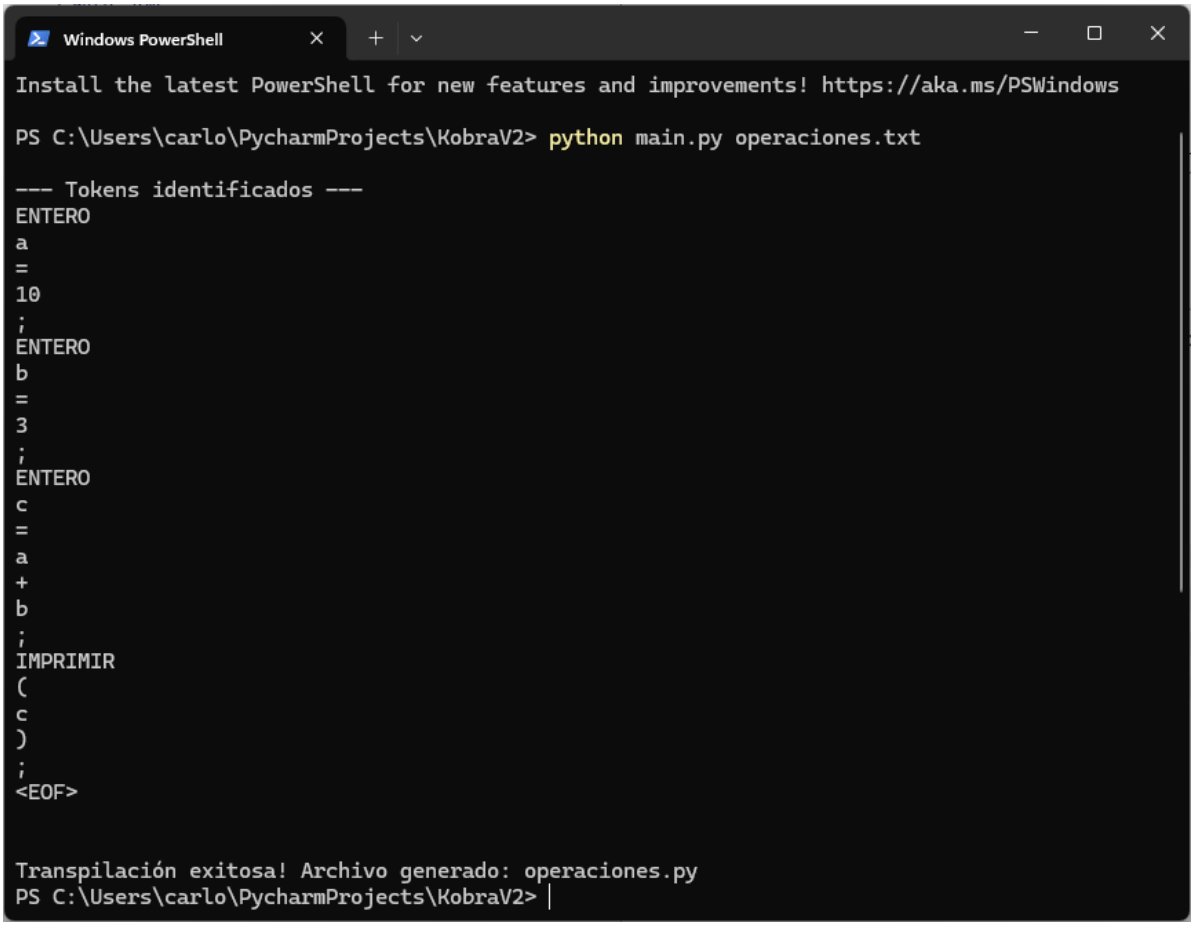
```
    ENTERO a = 10;
    ENTERO b = 3;
    ENTERO c = a + b;

    IMPRIMIR(c);
```

Comando de ejecución:

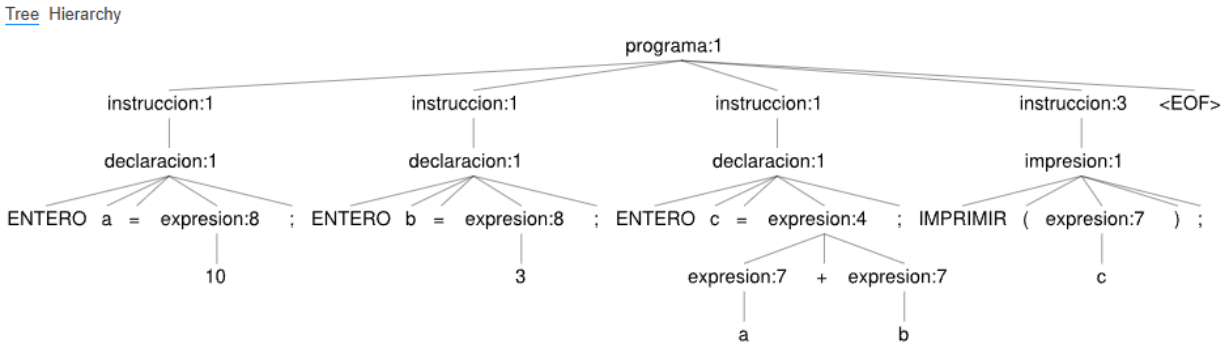
```
python main.py operaciones.txt
```



Tokens identificados durante el análisis léxico (Lexer):



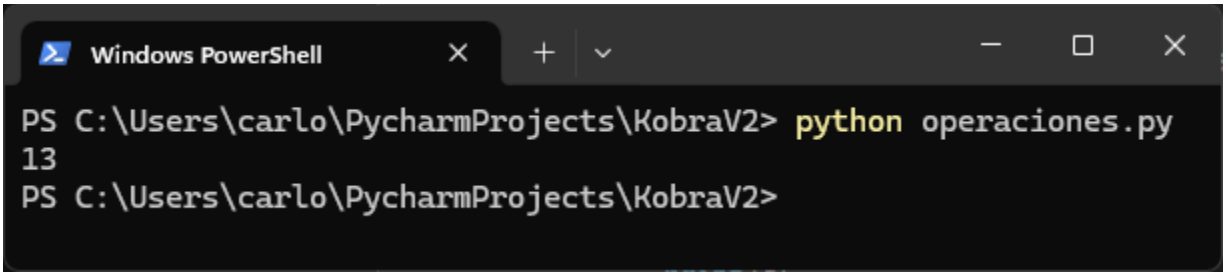
Árbol sintáctico generado mostrando la estructura del programa y las instrucciones construidas:



Archivo de salida generado:
operaciones.py

```python
# -*- coding: utf-8 -*-
# Codigo generado desde KobraV2
import sys


def _kobra_div(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a // b
    return a / b

a = 10
b = 3
c = a + b
print(c)
```

Prueba de funcionamiento del archivo de salida en Python:



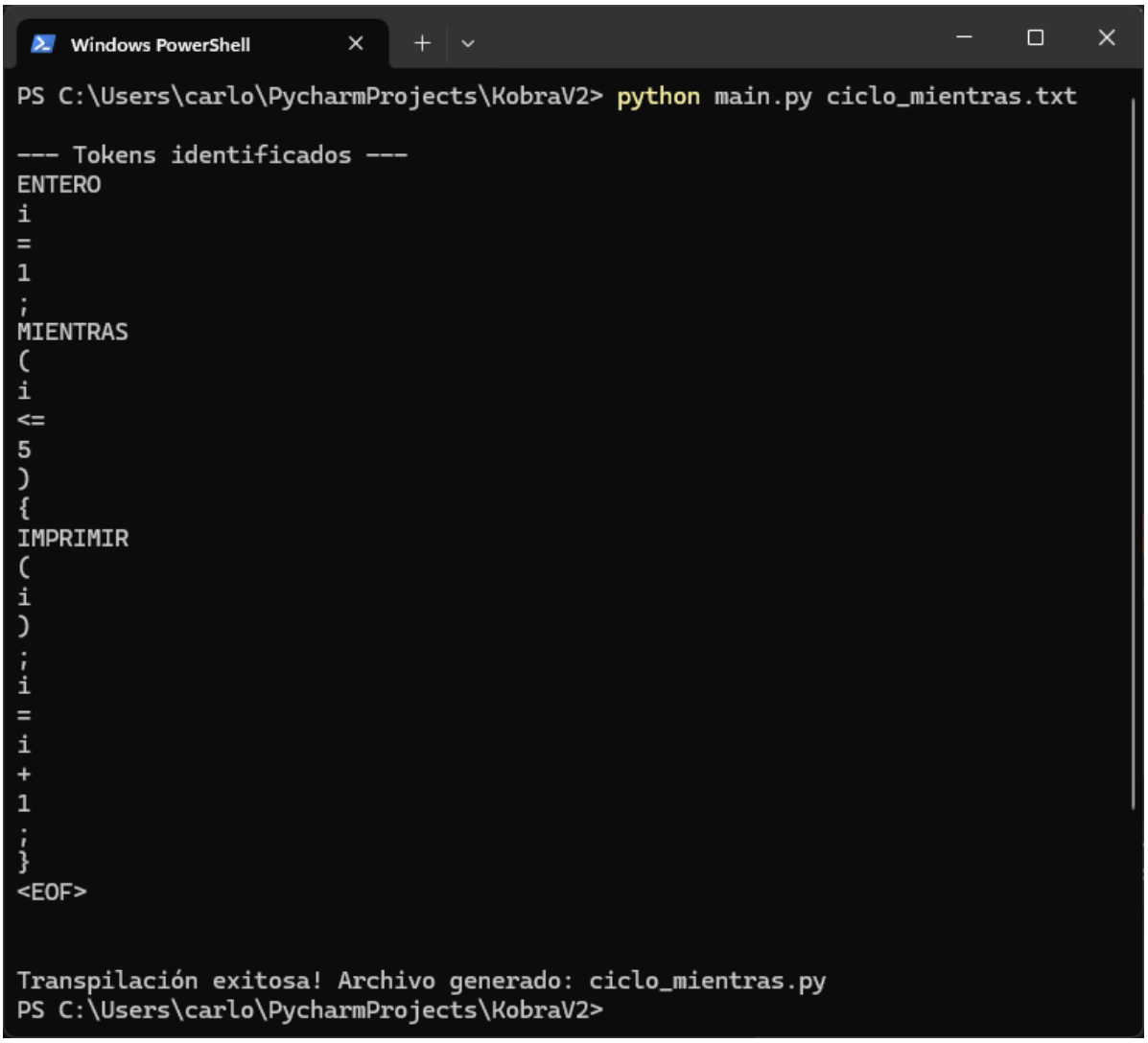
Ejemplo 2:

Entrada:

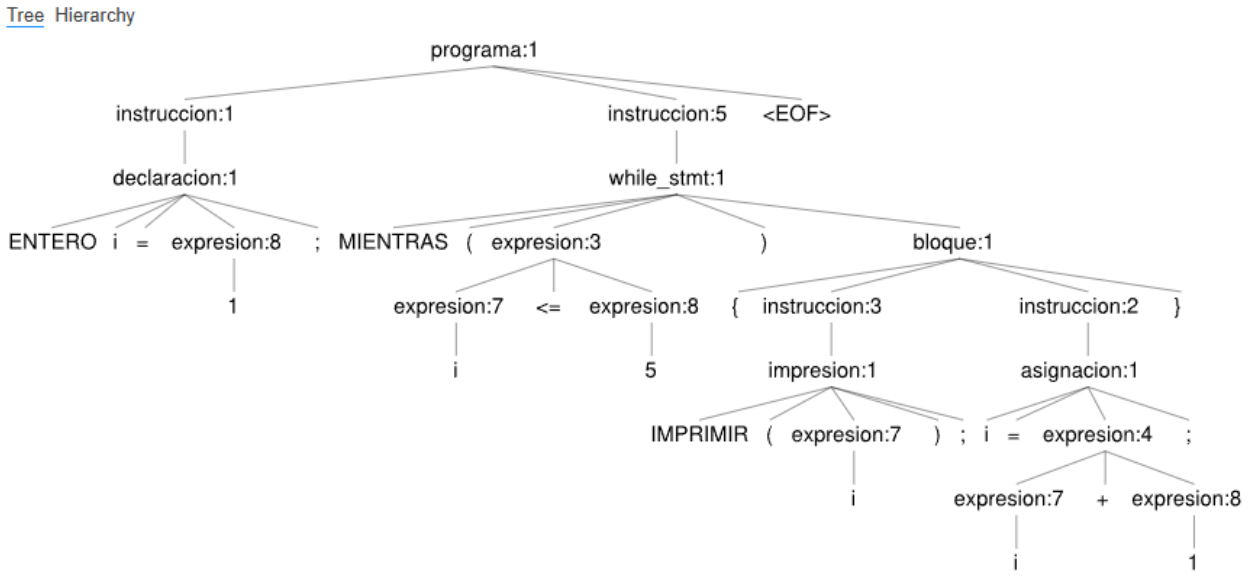
ciclo_mientras.txt

```
ENTERO i = 1;


MIENTRAS (i <= 5) {
    IMPRIMIR(i);
    i = i + 1;
}
```

Ejecución de comando con tokens identificados:



Árbol sintáctico generado:



Archivo de salida:
ciclo_mientras.py

```
# -*- coding: utf-8 -*-
# Codigo generado desde KobraV2
import sys


def _kobra_div(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a // b
    return a / b

i = 1
while i <= 5:
    print(i)
    i = i + 1
```

Prueba de funcionamiento:

```
Windows PowerShell
PS C:\Users\carlo\PycharmProjects\KobraV2> python ciclo_mientras.py
1
2
3
4
5
PS C:\Users\carlo\PycharmProjects\KobraV2>
```

Ejemplo 3:
Entrada:
cuenta_restaurante.txt

```
// --- CALCULADORA DE CUENTA RESTAURANTE ---
FLOTANTE costoComida = 450.50;
FLOTANTE costoBebidas = 120.00;
FLOTANTE tasaPropina = 0.15;
ENTERO personas = 3;
FLOTANTE total = 0.0;
FLOTANTE propina = 0.0;
FLOTANTE totalFinal = 0.0;
FLOTANTE porPersona = 0.0;

total = costoComida + costoBebidas;

IMPRIMIR("--- TICKET DE KOBRA RESTAURANT ---");
IMPRIMIR("Subtotal comida y bebida:");
IMPRIMIR(total);

// Logica de propina alta si el consumo fue alto
SI (total > 500.0) {
    IMPRIMIR("Consumo alto detectado. Aplicando propina sugerida
del 15%");
    propina = total * tasaPropina;
} SINO {
    IMPRIMIR("Consumo estandar. Propina fija de 50 pesos.");
    propina = 50.0;
}
```

```
totalFinal = total + propina;

IMPRIMIR("Propina calculada:");
IMPRIMIR(propina);
IMPRIMIR("TOTAL A PAGAR:");
IMPRIMIR(totalFinal);

porPersona = totalFinal / personas;

IMPRIMIR("Monto a pagar por persona:");
IMPRIMIR(porPersona);

IMPRIMIR("Imprimiendo recibos individuales...");
ENTERO i = 0;
PARA (i = 0; i < personas; i = i + 1) {
    IMPRIMIR("Recibo impreso.");
}
```

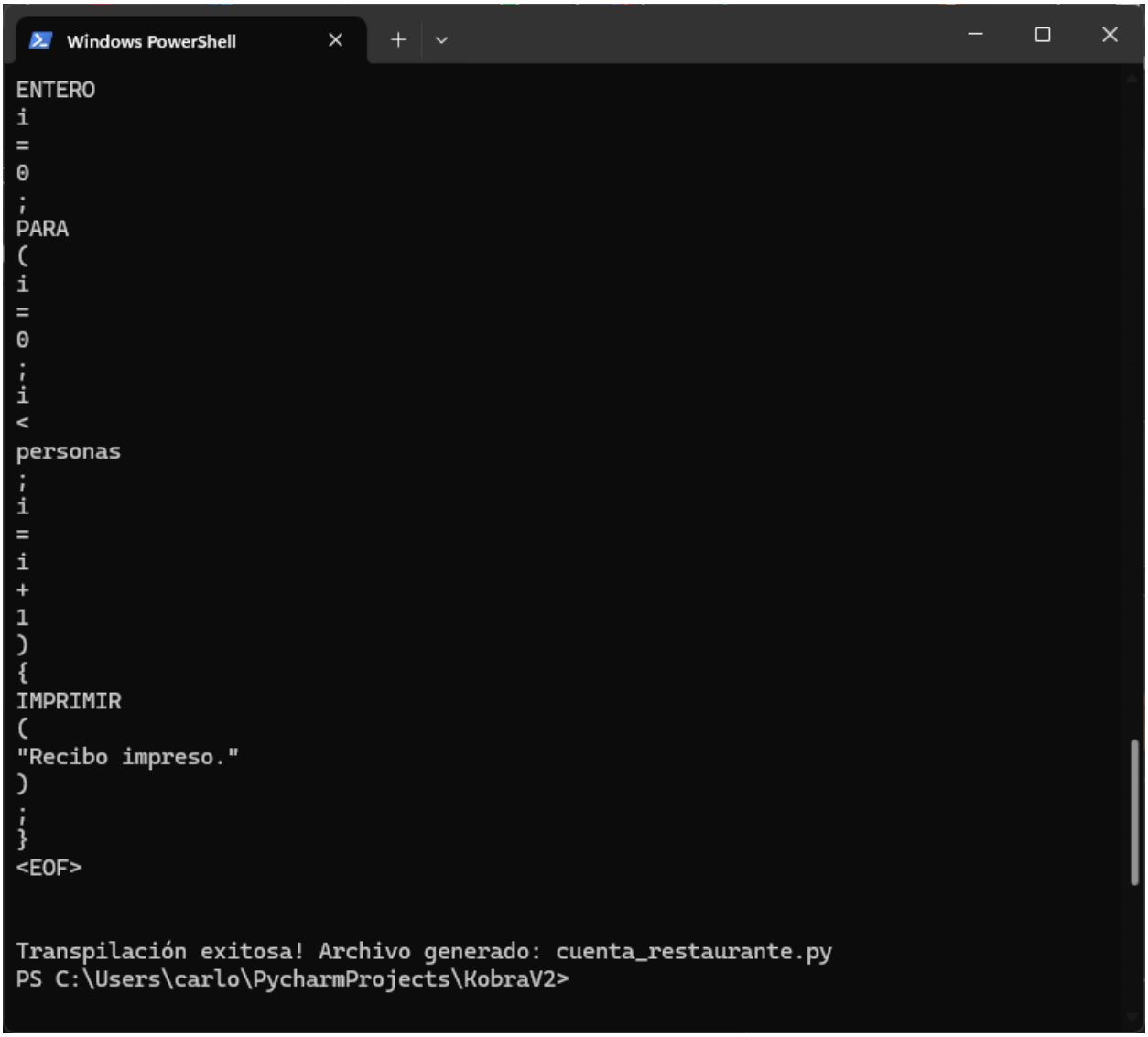
Ejecución de comando con tokens identificados:

Windows PowerShell

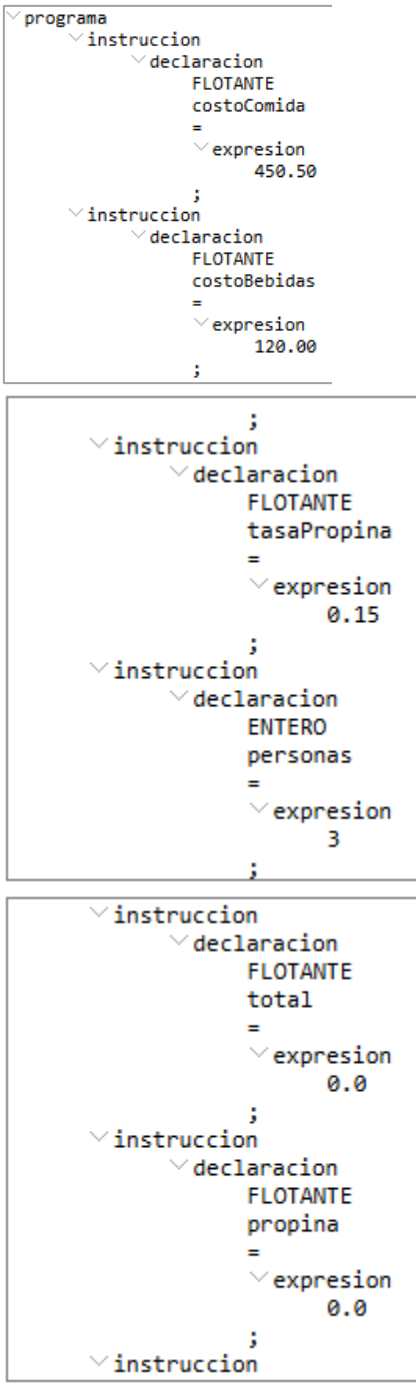
```
costoBebidas
;
IMPRIMIR
(
"--- TICKET DE KOBRA RESTAURANT ---"
)
;
IMPRIMIR
(
"Subtotal comida y bebida:"
)
;
IMPRIMIR
(
total
)
;
SI
(
total
>
500.0
)
{
IMPRIMIR
(
"Consumo alto detectado. Aplicando propina sugerida del 15%"
)
;
propina
=
total
*
tasaPropina
;
}
SINO
{
IMPRIMIR
(
"Consumo estandar. Propina fija de 50 pesos."
)
;
propina
=
50.0
;
}
totalFinal
=
total
```

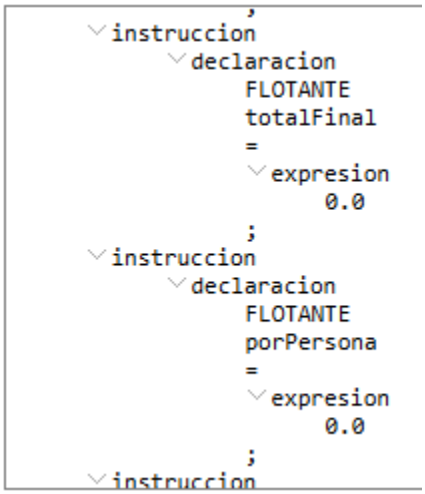
Windows PowerShell

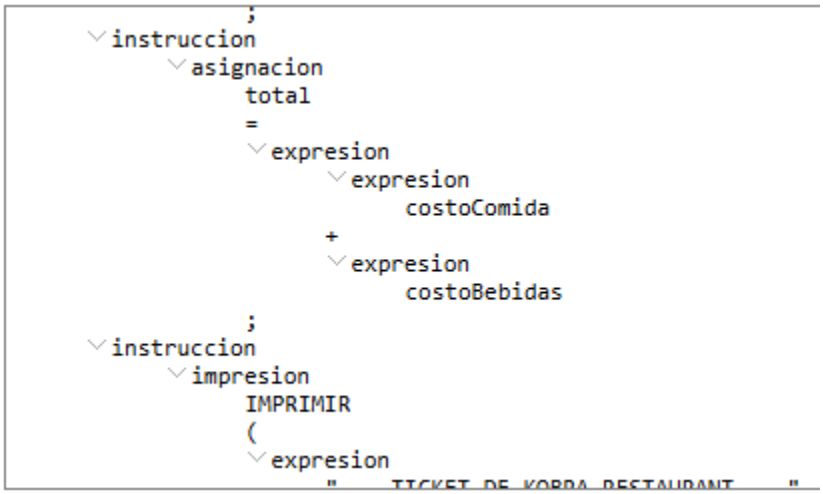
```
totalFinal
=
total
+
propina
;
IMPRIMIR
(
"Propina calculada:"
)
;
IMPRIMIR
(
propina
)
;
IMPRIMIR
(
"TOTAL A PAGAR:"
)
;
IMPRIMIR
(
totalFinal
)
;
porPersona
=
totalFinal
/
personas
;
IMPRIMIR
(
"Monto a pagar por persona:"
)
;
IMPRIMIR
(
porPersona
)
;
IMPRIMIR
(
"Imprimiendo recibos individuales..."
)
;
ENTERO
i
=
0
```

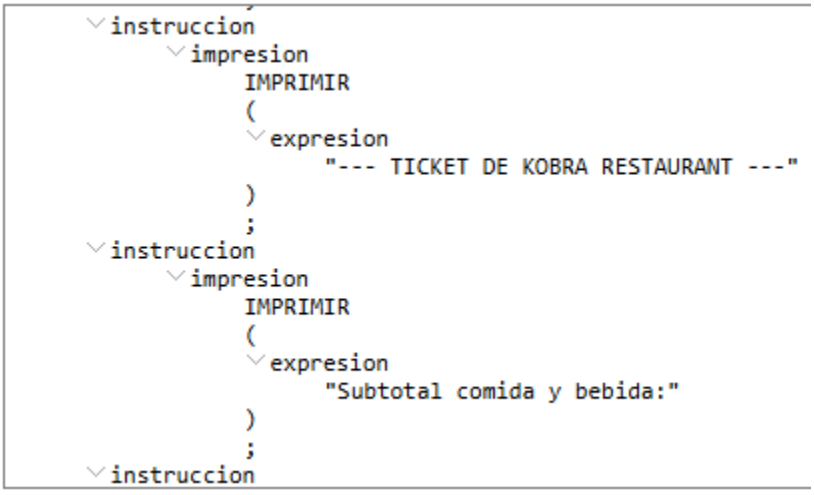

Windows PowerShell
ENTERO
i
=
0
;
PARA
(
i
=
0
;
i
<
personas
;
i
=
i
+
1
)
{
IMPRIMIR
(
"Recibo impreso."
)
;
}
<EOF>
Transpilación exitosa! Archivo generado: cuenta_restaurante.py
PS C:\Users\carlo\PycharmProjects\KobraV2>

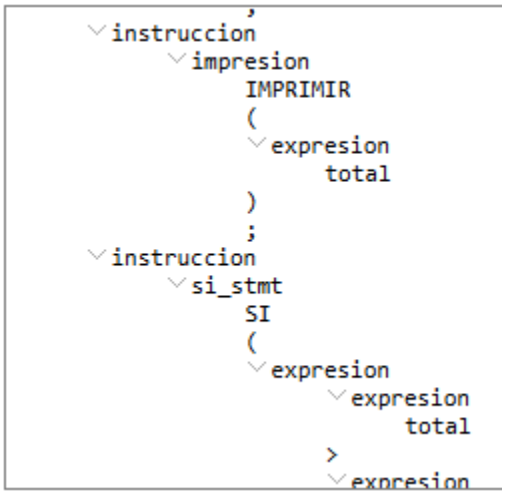
Árbol sintáctico generado representado mediante jerarquía de instrucciones:

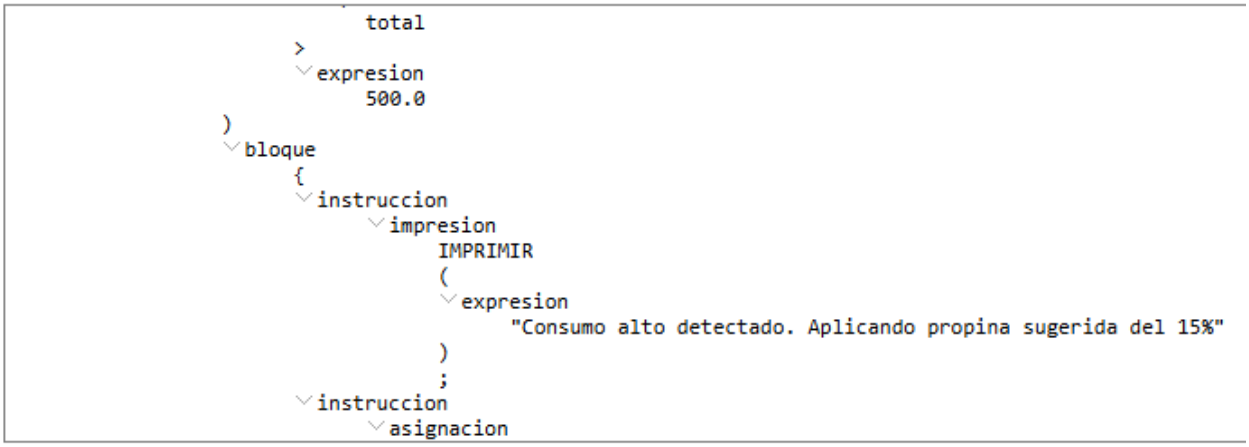


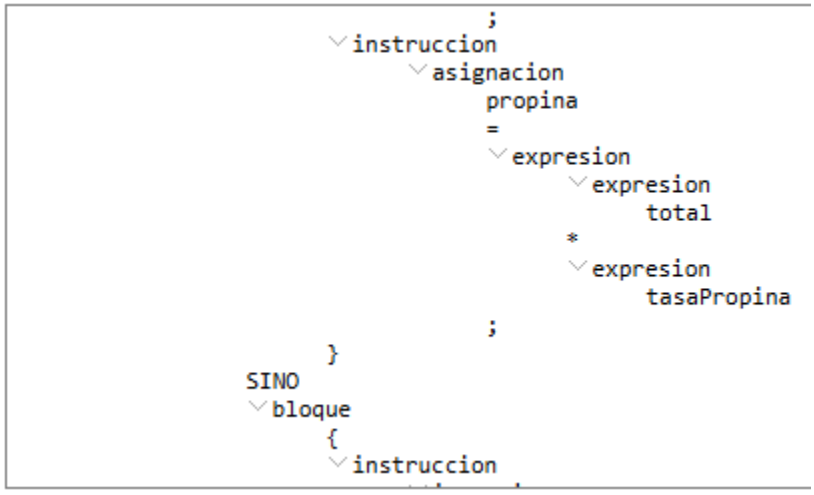

;
instruccion
declaracion
FLOTANTE
totalFinal
=
expresion
0.0
;
instruccion
declaracion
FLOTANTE
porPersona
=
expresion
0.0
;
instruccion


;
instruccion
asignacion
total
=
expresion
expresion
costoComida
+
expresion
costoBebidas
;
instruccion
impresion
IMPRIMIR
(
expresion


instruccion
impresion
IMPRIMIR
(
expresion
"--- TICKET DE KOBRA RESTAURANT ---"
)
;
instruccion
impresion
IMPRIMIR
(
expresion
"Subtotal comida y bebida:"
)
;
instruccion


;
instruccion
impresion
IMPRIMIR
(
expresion
total
)
;
instruccion
si_stmt
SI
(
expresion
expresion
total
>


total
>
expresion
500.0
)
bloque
{
instruccion
impresion
IMPRIMIR
(
expresion
"Consumo alto detectado. Aplicando propina sugerida del 15%"
)
;
instruccion
asignacion

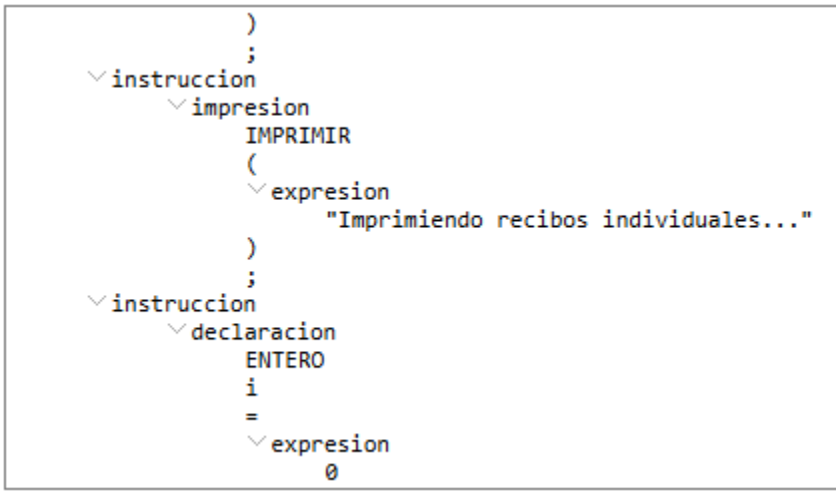

;
instruccion
asignacion
propina
=
expresion
expresion
total
*
expresion
tasaPropina
;
}
SINO
bloque
{
instruccion

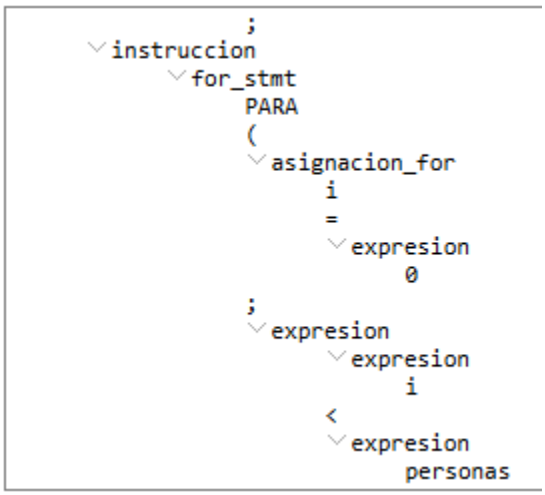
```
                    ;
                }
            SINO
            bloque
                {
                instruccion
                    impresion
                        IMPRIMIR
                        (
                        expresion
                            "Consumo estandar. Propina fija de 50 pesos."
                        )
                        ;
                instruccion
                    asignacion
                        propina
                        =
                        expresion
                        ;
                instruccion
                    asignacion
                        propina
                        =
                        expresion
                            50.0
                        ;
                }
    instruccion
        asignacion
            totalFinal
            =
            expresion
                expresion
                    total
                +
                expresion
                    total
                +
                expresion
                    propina
            ;
    instruccion
        impresion
            IMPRIMIR
            (
            expresion
                "Propina calculada:"
            )
            ;
    instruccion
        impresion
            IMPRIMIR
```

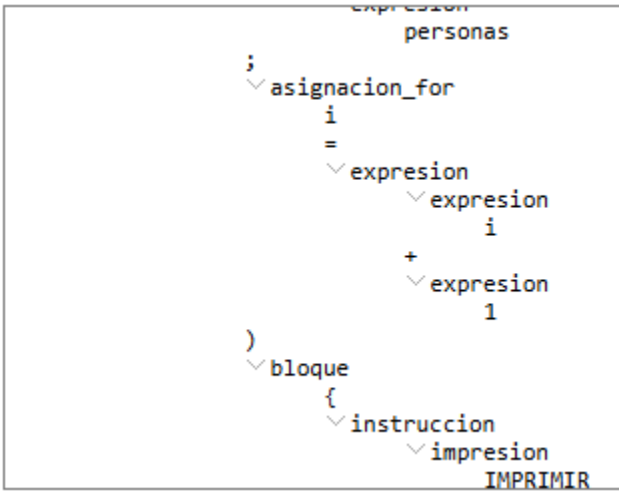
```
            IMPRIMIR
            (
            expresion
                propina
            )
            ;
    instruccion
        impresion
            IMPRIMIR
            (
            expresion
                "TOTAL A PAGAR:"
            )
            ;
    instruccion
        impresion
            IMPRIMIR
```

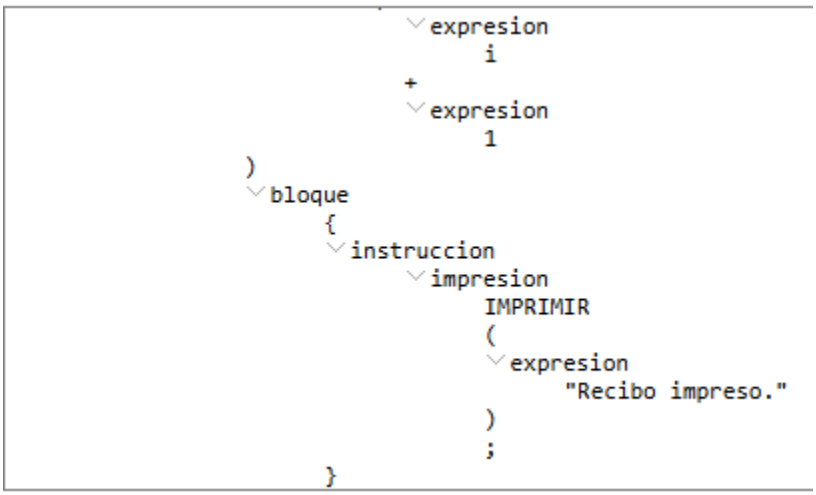
```
        impresion
            IMPRIMIR
            (
            expresion
                totalFinal
            )
            ;
    instruccion
        asignacion
            porPersona
            =
            expresion
                expresion
                    totalFinal
                /
                expresion
                    personas
            ;
```

```
    instruccion
        impresion
            IMPRIMIR
            (
            expresion
                "Monto a pagar por persona:"
            )
            ;
    instruccion
        impresion
            IMPRIMIR
            (
            expresion
                porPersona
            )
            ;
    instruccion
```


)
;
instruccion
impresion
IMPRIMIR
(
expresion
"Imprimiendo recibos individuales..."
)
;
instruccion
declaracion
ENTERO
i
=
expresion
0


;
instruccion
for_stmt
PARA
(
asignacion_for
i
=
expresion
0
;
expresion
expresion
i
<
expresion
personas


personas
;
asignacion_for
i
=
expresion
expresion
i
+
expresion
1
)
bloque
{
instruccion
impresion
IMPRIMIR

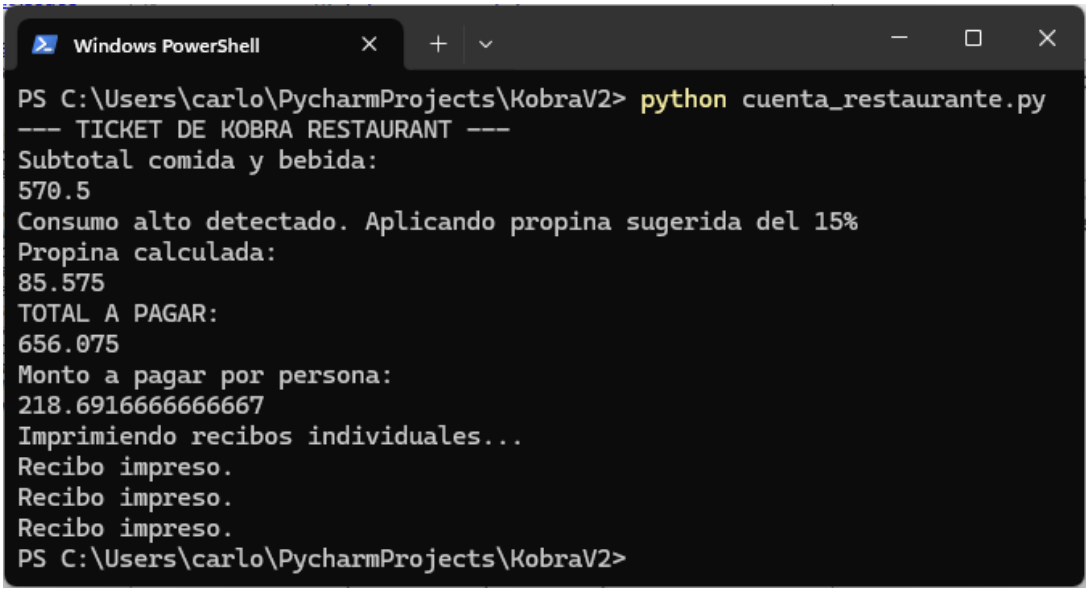


Archivo de salida:
cuenta_restaurante.py

```
# -*- coding: utf-8 -*-
# Codigo generado desde KobraV2
import sys

def _kobra_div(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a // b
    return a / b

costoComida = 450.50
costoBebidas = 120.00
tasaPropina = 0.15
personas = 3
total = 0.0
propina = 0.0
totalFinal = 0.0
porPersona = 0.0
total = costoComida + costoBebidas
print("--- TICKET DE KOBRA RESTAURANT ---")
print("Subtotal comida y bebida:")
print(total)
if total > 500.0:
    print("Consumo alto detectado. Aplicando propina sugerida del 15%")
    propina = total * tasaPropina
else:
    print("Consumo estandar. Propina fija de 50 pesos.")
    propina = 50.0
totalFinal = total + propina
print("Propina calculada:")
print(propina)
print("TOTAL A PAGAR:")
```

```
print(totalFinal)
porPersona = _kobra_div(totalFinal, personas)
print("Monto a pagar por persona:")
print(porPersona)
print("Imprimiendo recibos individuales...")
i = 0
i = 0
while i < personas:
    print("Recibo impreso.")
    i = i + 1
```

Prueba de funcionamiento:



Manejo de errores

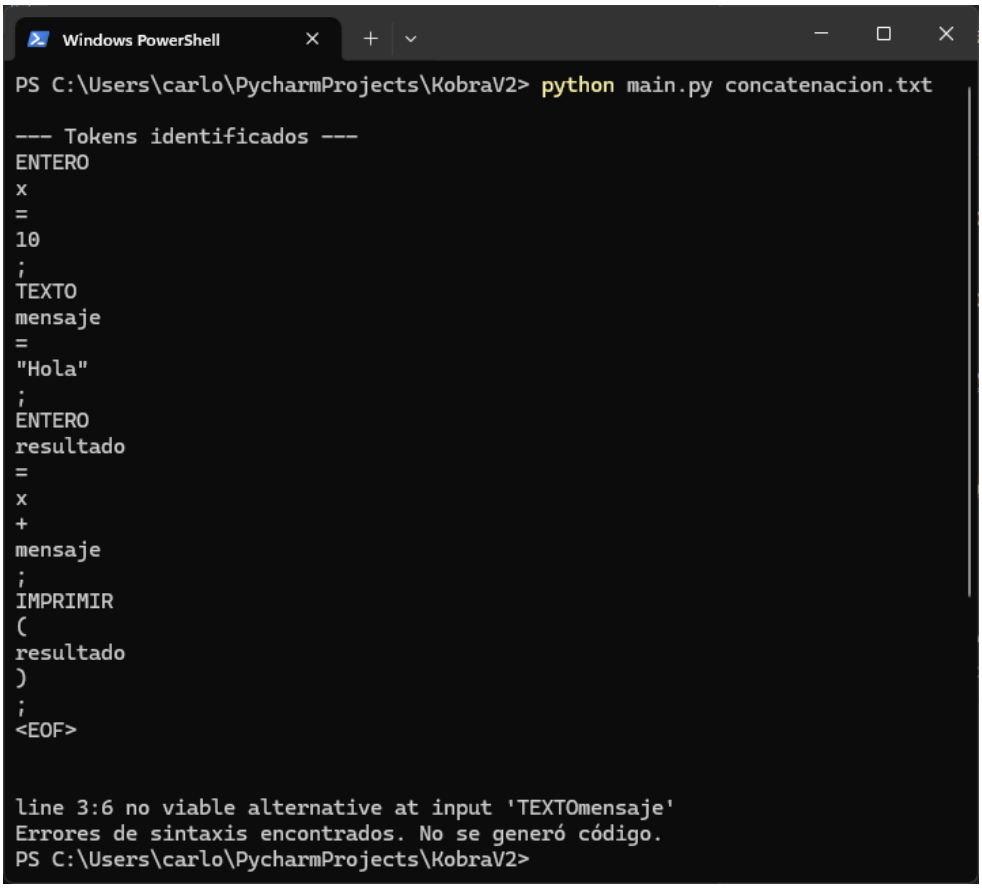
Ejemplo 1: Operaciones con tipos de datos incompatibles

Entrada:

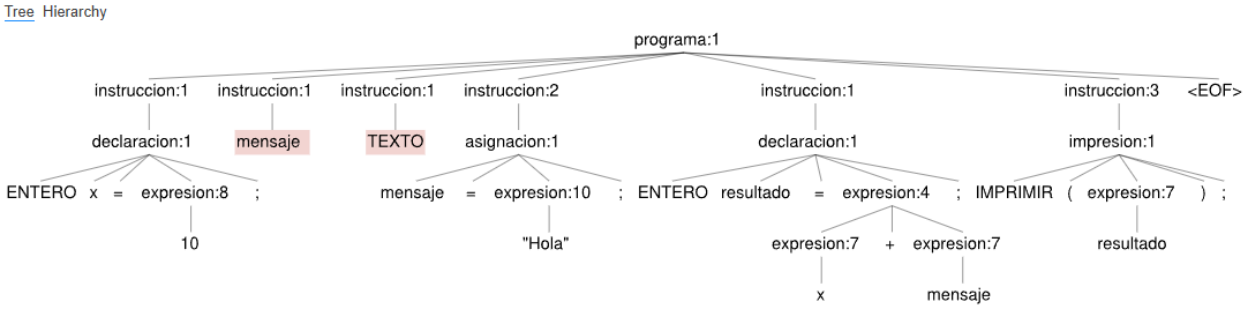
concatenación.txt

```
ENTERO x = 10;
TEXTO mensaje = "Hola";
ENTERO resultado = x + mensaje;

IMPRIMIR(resultado);
```

Ejecución de comando:



Árbol sintáctico correspondiente a la entrada mostrando la falta de acción semántica correspondiente a la suma de variables tipo entero:



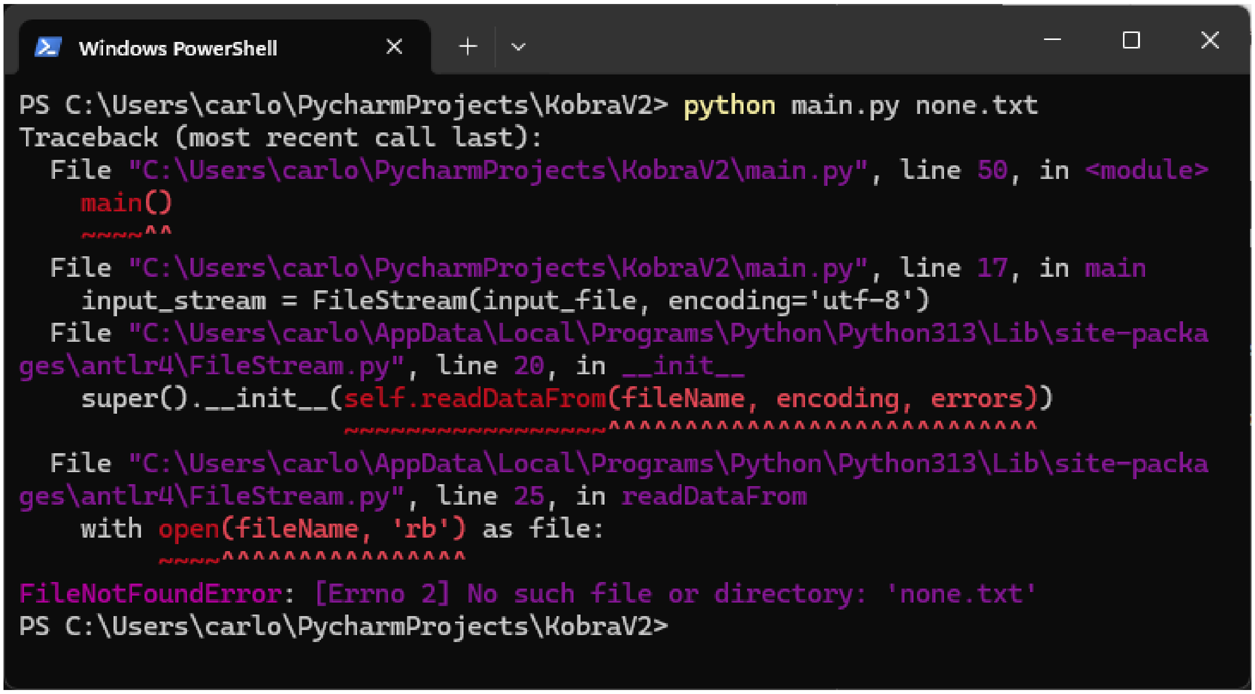
Ejemplo 2: Archivo de entrada inexistente

Entrada:

none.txt

NOTA: Este archivo es inexistente al momento de la ejecución del archivo main.py.

Ejecución del programa main.py:



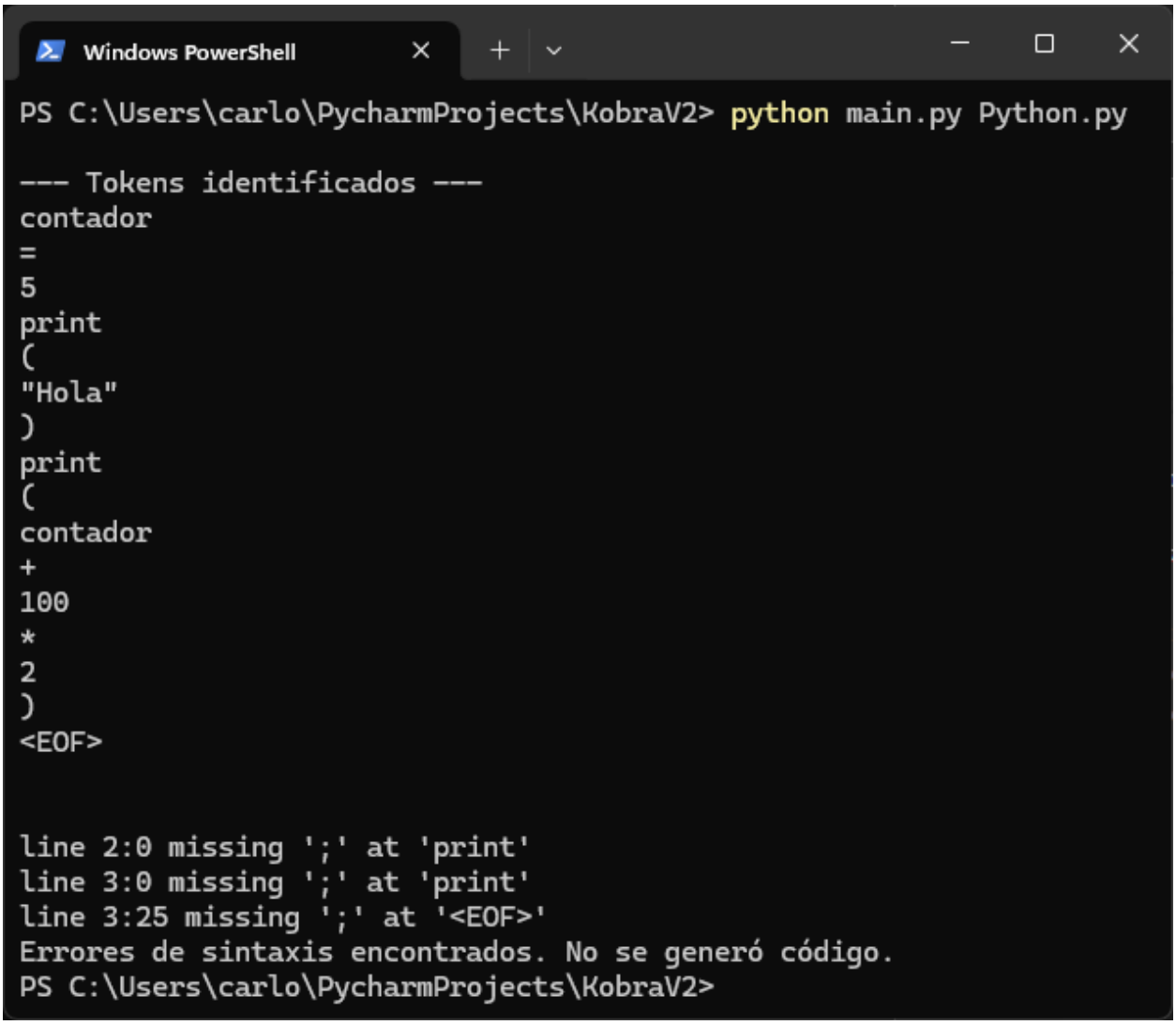
Ejemplo 3: Lenguaje de entrada incompatible

Entrada:

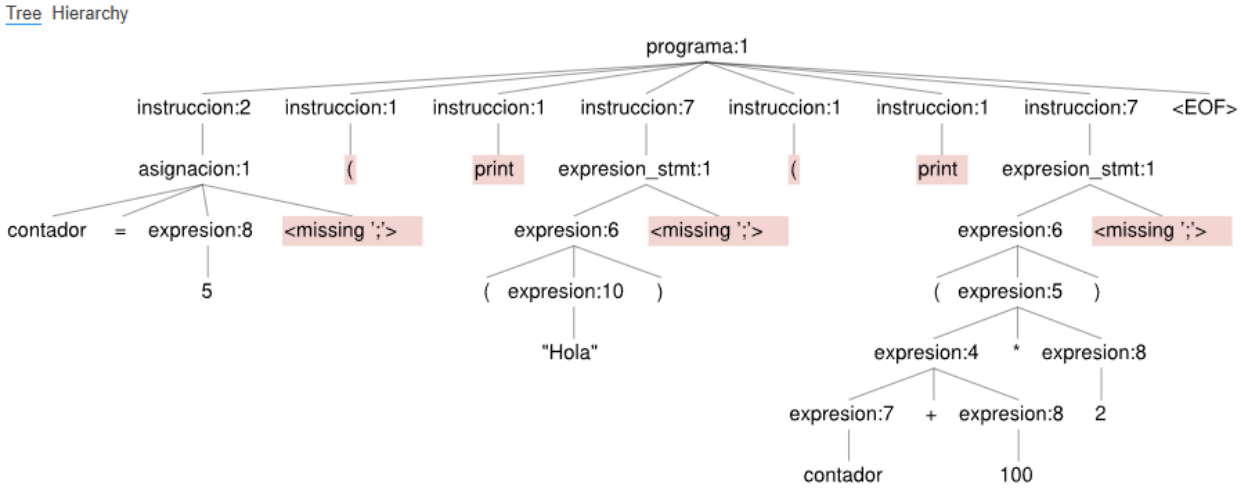
Python.py

```
contador = 5
print("Hola")
print(contador + 100 * 2)
```

Ejecución del programa main.py:



Árbol que se hubiera generado con el código de entrada, mostrando claramente los errores sintácticos por las faltas de tokens para marcar el fin de una instrucción y palabras claves incompatibles (“print”):



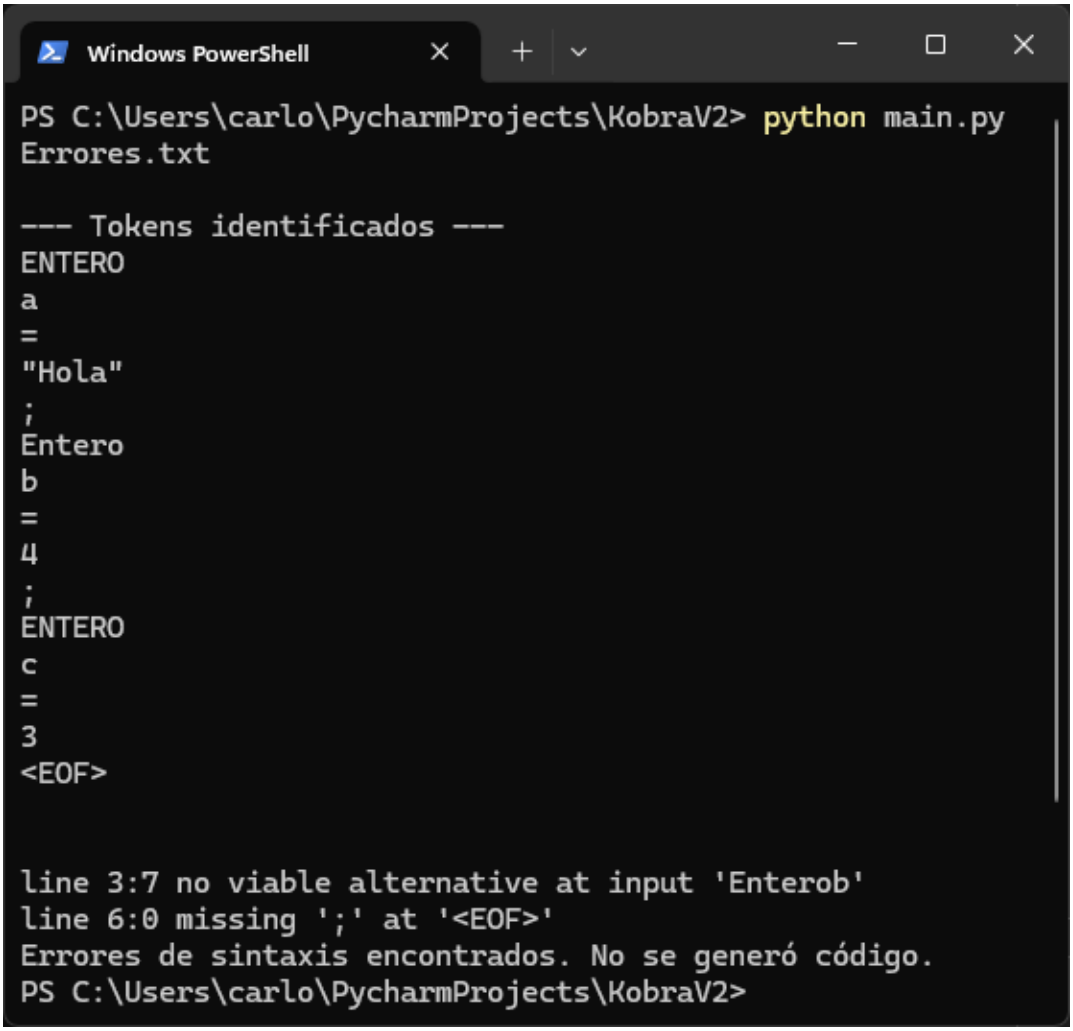
Ejemplo 4: Errores sintácticos
Entrada:
Errores.txt

ENTERO a = "Hola";

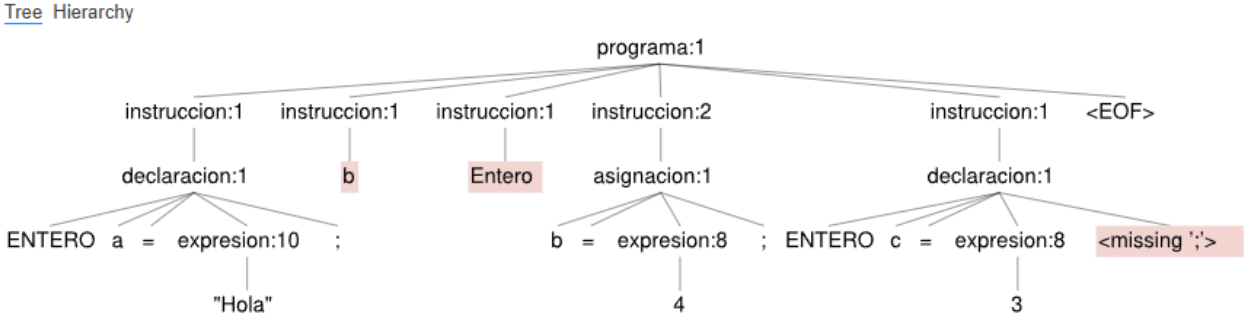
Entero b = 4;

ENTERO c = 3

Ejecución del programa main.py mostrando los errores en la estructura de las instrucciones detectados:



Árbol correspondiente que se hubiera generado con el código de entrada:



4. Referencia técnica

Resumen del lenguaje

KobraV2 se caracteriza por los siguientes principios de diseño:

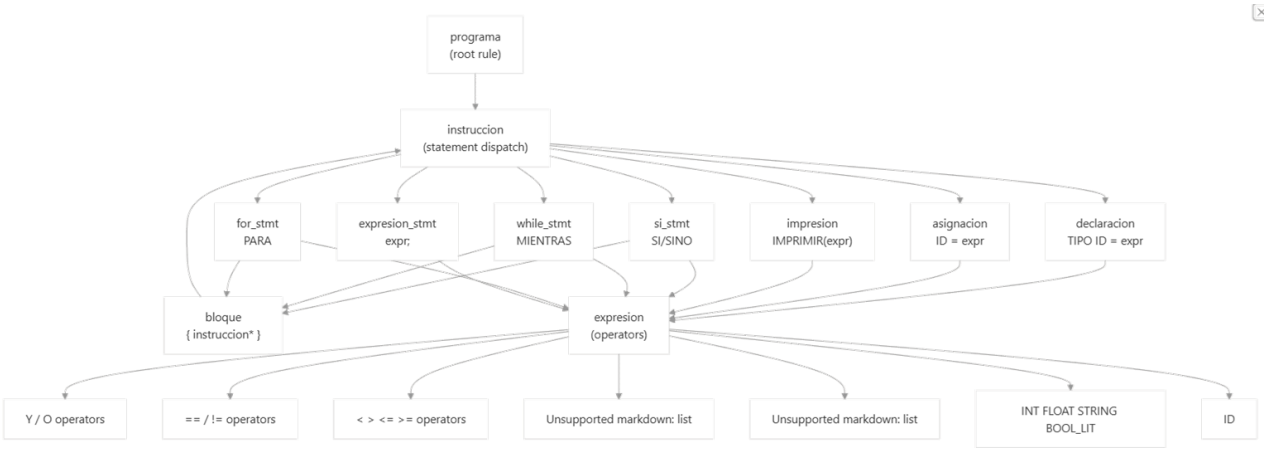
Principio	Requerimiento
Español como prioridad	Todas las palabras clave, nombres de tipo y literales booleanos usan términos en español.
Enfoque	Sintaxis simple y explicita, adecuada para enseñar los fundamentos de programación.
Sintaxis estilo C	Estructura familiar con llaves `{}`, punto y coma `;` y paréntesis `()` para el flujo de control.
Tipado estático	Declaraciones de tipo explícitas usando la palabra clave `TIPO`.
Paradigma imperativo	Ejecución secuencial con estado mutable y flujo de control.

Características clave del lenguaje

- Cuatro tipos primitivos: ENTERO (integer), FLOTANTE (float), CADENA (string), BOOLEANO (boolean).
- Tres estructuras de control de flujo: SI/SINO (if/else), MIENTRAS (while), PARA (for).
- Cinco categorías de operadores: Aritméticos, relacionales, de igualdad, lógicos y de asignación.
- Entrada/Salida (E/S) básica: Sentencia IMPRIMIR para la salida de datos.
- Operadores en español: Y (and), O (or) para la lógica booleana.
- Soporte de comentarios: Comentarios de una sola línea estilo C (//).

Arquitectura del lenguaje

El lenguaje KobraV2 está estructurado jerárquicamente, donde la gramática define cómo los tokens se combinan para formar expresiones, sentencias y programas completos.



Estructura del programa

Un programa KobraV2 consta de una secuencia de sentencias (instruccion*) terminada por EOF, tal como lo define la regla programa.

```
programa: instruccion* EOF;
```

Tipos de sentencias

- **declaración.** Declaración de variable con inicialización opcional.

  ```
  declaracion: TIPO ID ('=' expresion)? ';';
  ```

- **asignación.** Asignación de variable

  ```
  asignacion: ID '=' expresion ';';
  ```

- **impresión.** Sentencia de salida (output).

  ```
  impresion: 'IMPRIMIR' '(' expresion ')' ';';
  ```

- **si_stmt**. Ejecución condicional.

  ```
  si_stmt: 'SI' '(' expresion ')' bloque ('SINO' bloque)?;
  ```

- **while_stmt.** Bucle de pre-condición (ciclo while)

  ```
  while_stmt: 'MIENTRAS' '(' expresion ')' bloque;
  ```

- **for_stmt.** Bucle contado (ciclo for)

  ```
  for_stmt: 'PARA' '(' asignacion_for ';' expresion ';'
  asignacion_for ')' bloque;
  ```

- **expression_stmt.** Evaluación de expresión con efectos secundarios.

```
expresion_stmt: expresion ';';
```

Estructura de bloques

La regla `bloque` define los bloques de código:

```
bloque: '{' instruccion* '}';
```

Los bloques:

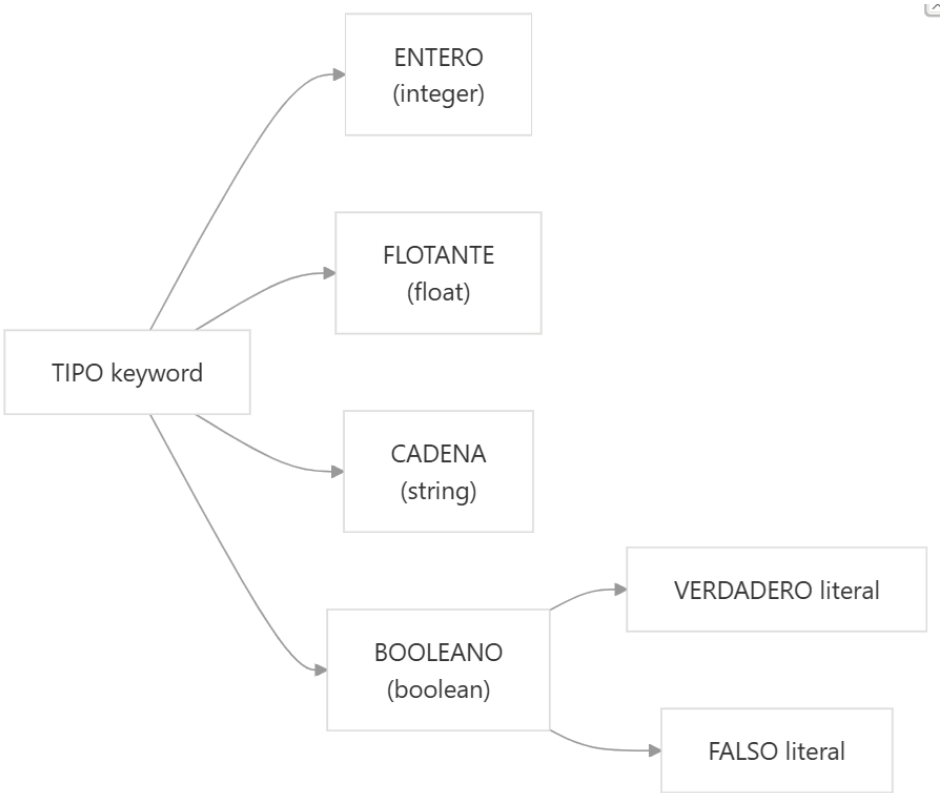
- Contienen cero o más instrucciones (sentencias).
- Están delimitados por llaves `{}`.
- Son obligatorios para las estructuras de control de flujo.

Descripción General de las Características del Lenguaje

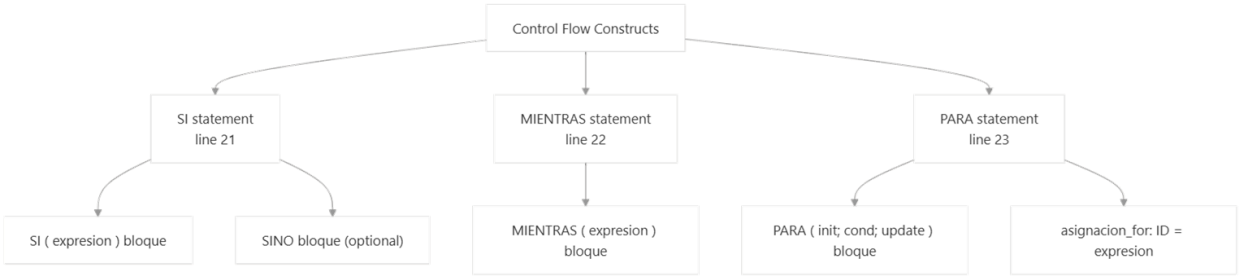
Sistema de Tipos

KobraV2 proporciona cuatro tipos primitivos declarados mediante el token `TIPO`.

```
TIPO ID ('=' expresion)? ';'
```



Control de flujo



Reglas gramaticales de Flujo de Control

Estructura	Sintaxis
Condicional	`'SI' '(' expresion ')' bloque ('SINO' bloque)?;`
Ciclo MIENTRAS (while)	`'MIENTRAS' '(' expresion ')' bloque;`
Ciclo PARA (for)	`'PARA' '(' asignacion_for ';' expresion ';' asignacion_for ')' bloque;`

La regla `for_stmt` utiliza una regla especializada `asignacion_for` que omite el punto y coma de terminación para su uso dentro del encabezado del ciclo `for`.

```
asignacion_for: ID '=' expresion;
```

Expresiones y operadores

La regla `expresión` define una jerarquía de precedencia utilizando la sintaxis recursiva por la izquierda de ANTLR.

Precedencia	Operadores	Nombre Alternativo	Etiqueta Gramatical
1 (más baja)	`Y, O`	OpLogica	Lógica
2	`==, !=`	OpIgualdad	Igualdad
3	`<, >, <=, >=`	OpRelacional	Relacional
4	`+, -`	OpAditiva	Aditiva

5 (más alta)	`*, /`	OpMultiplicativa	Multiplicativa

Los nombres alternativos se utilizan como etiquetas en la gramática para el patrón visitante de ANTLR.

Elementos Léxicos

Palabras clave

Categoría	Palabra Clave	Propósito
Tipos	`ENTERO, FLOTANTE, CADENA, BOOLENO`	Declaración de tipo
Control	`SI, SINO, MIENTRAS, PARA`	Flujo de control
I/O	`IMPRIMIR`	Salida (output)
Lógica	`Y, O`	Operadores booleanos
Marcador de tipo	`TIPO`	Introduce declaraciones de tipo

Literales

Token	Patrón
INT	`[0-9]+`
FLOAT	`[0-9]+ '.' [0-9]+`
STRING	`'"' ~["]* '"'`
BOOL_LIT	`'VERDADERO' | 'FALSO'`

Identificadores

El token ID define los identificadores válidos:

```
ID: [a-zA-Z_][a-zA-Z0-9_]*;
```

Reglas:

- Debe comenzar con una letra (a-z, A-Z) o un guion bajo (_).

- Puede contener letras, dígitos y guiones bajos.
- Distingue entre mayúsculas y minúsculas.
- No puede ser una palabra clave reservada.

Convenciones de Sintaxis

Terminación de Sentencias

Todas las sentencias, excepto las estructuras de flujo de control deben terminar con un punto y coma (`;`).

- `declaracion` termina con `;`
- `asignacion` termina con `;`
- `impresion` termina con `;`
- `expresion_stmt` termina con `;`

Las estructuras de flujo de control utilizan bloques y no requieren puntos y coma.

Delimitadores de Bloque

Todas las estructuras de control de flujo requieren bloques entre llaves:

```
SI (x > 0) {
    IMPRIMIR(x);
}
```

Los bloques de una sola sentencia también requieren llaves:

```
MIENTRAS (contador < 10) {
    contador = contador + 1;
}
```

Paréntesis en Expresiones

Las condiciones de flujo de control deben estar entre paréntesis:

- `si_stmt: SI '(' expresion ')' bloque`

- while_stmt: MIENTRAS '(' expresion ')' bloque
- for_stmt: PARA '(' ... ')' bloque

5. Código fuente

KobraV2.g4

```
grammar KobraV2;

programa: instruccion* EOF;

instruccion
    : declaracion
    | asignacion
    | impresion
    | si_stmt
    | while_stmt
    | for_stmt
    | expresion_stmt
    ;

bloque: '{' instruccion* '}';

declaracion: TIPO ID ('=' expresion)? ';';
asignacion: ID '=' expresion ';';
impresion: 'IMPRIMIR' '(' expresion ')' ';';

si_stmt: 'SI' '(' expresion ')' bloque ('SINO' bloque)?;
while_stmt: 'MIENTRAS' '(' expresion ')' bloque;
for_stmt: 'PARA' '(' asignacion_for ';' expresion ';' asignacion_for
')' bloque;
asignacion_for: ID '=' expresion;
expresion_stmt: expresion ';';

expresion
    : expresion op=('Y' | 'O') expresion                # OpLogica
    | expresion op=('==' | '!=') expresion              # OpIgualdad
    | expresion op=('<' | '>' | '<=' | '>=') expresion  #
```

```
OpRelacional
    | expresion op=('+' | '-') expresion                # OpAditiva
    | expresion op=('*' | '/') expresion                #
OpMultiplicativa
    | '(' expresion ')'                                 # Parentesis
    | ID                                                # Variable
    | INT                                               # Entero
    | FLOAT                                             # Flotante
    | STRING                                            # Cadena
    | BOOL_LIT                                          # Booleano
    ;

// --- LEXER ---

// CAMBIO: Tipos en Español
TIPO: 'ENTERO' | 'FLOTANTE' | 'CADENA' | 'BOOLEANO';

// CAMBIO: Booleanos en Español
BOOL_LIT: 'VERDADERO' | 'FALSO';

SI: 'SI';
SINO: 'SINO';
MIENTRAS: 'MIENTRAS';
PARA: 'PARA';
IMPRIMIR: 'IMPRIMIR';

Y: 'Y';
O: 'O';

ID: [a-zA-Z_][a-zA-Z0-9_]*;
INT: [0-9]+;
FLOAT: [0-9]+ '.' [0-9]+;
STRING: '"' ~["]* '"';
COMENTARIO: '//' ~[\r\n]* -> skip;
WS: [ \t\r\n]+ -> skip;
}}
```

KobraCompiler.py

```
import sys
from antlr4 import *
from KobraV2Lexer import KobraV2Lexer
from KobraV2Parser import KobraV2Parser
from KobraV2Visitor import KobraV2Visitor

class KobraToPythonVisitor(KobraV2Visitor):
    def __init__(self):
        self.indent_level = 0
        self.output = ""

    def add_line(self, line):
        # Agrega indentación y la linea
        self.output += ("    " * self.indent_level) + line + "\n"

    def visitPrograma(self, ctx):
        self.add_line("# -*- coding: utf-8 -*-")

        self.add_line("# Codigo generado desde KobraV2")
        self.add_line("import sys")
        self.add_line("")

        # --- HELPER PARA DIVISION ---
        # Inyectamos esta funcion para manejar division entera vs
flotante
        helper_code = """
def _kobra_div(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a // b
    return a / b
"""
        # Añadimos el helper al output sin indentar (nivel 0)
        self.output += helper_code + "\n"

        # Visitamos el resto de los hijos (las instrucciones)
```

```
        self.visitChildren(ctx)

        # --- ¡ESTA ES LA LINEA QUE FALTABA! ---
        return self.output

    def visitDeclaracion(self, ctx):
        # Kobra: INT x = 10; -> Py: x = 10
        # Ignoramos el tipo explicito en la traduccion simple (Python
es dinamico)
        var_name = ctx.ID().getText()
        if ctx.expresion():
            valor = self.visit(ctx.expresion())
            self.add_line(f"{var_name} = {valor}")
        else:
            # Si solo declara sin valor (INT x;), inicializamos en
None
            self.add_line(f"{var_name} = None")
        return None

    def visitAsignacion(self, ctx):
        var_name = ctx.ID().getText()
        valor = self.visit(ctx.expresion())
        self.add_line(f"{var_name} = {valor}")
        return None

    def visitImpresion(self, ctx):
        valor = self.visit(ctx.expresion())
        self.add_line(f"print({valor})")
        return None

    def visitSi_stmt(self, ctx):
        condicion = self.visit(ctx.expresion())
        self.add_line(f"if {condicion}:")

        # Visitamos el bloque VERDADERO
        self.visit(ctx.bloque(0))  # Primer bloque
```

```
        # Si hay SINO
        if ctx.bloque(1):
            self.add_line("else:")
            self.visit(ctx.bloque(1))
        return None

    def visitWhile_stmt(self, ctx):
        condicion = self.visit(ctx.expresion())
        self.add_line(f"while {condicion}:")
        self.visit(ctx.bloque())
        return None

    def visitFor_stmt(self, ctx):
        # Kobra: PARA (i=0; i<10; i=i+1) { ... }
        # Python: i=0; while i<10: ... i=i+1
        # Convertiremos el FOR de C en un WHILE de Python para
facilitar la traduccion

        # 1. Inicialización (se imprime antes del ciclo)
        init_stmt = ctx.asignacion_for(0)
        var_init = init_stmt.ID().getText()
        val_init = self.visit(init_stmt.expresion())
        self.add_line(f"{var_init} = {val_init}")

        # 2. Condición
        condicion = self.visit(ctx.expresion())
        self.add_line(f"while {condicion}:")

        # Aumentamos indentacion para el cuerpo
        self.indent_level += 1

        # 3. Cuerpo del loop (extraemos las instrucciones del bloque
manualmente)
        # El bloque tiene hijos, visitamos los hijos que sean
instrucciones
        bloque_ctx = ctx.bloque()
        for child in bloque_ctx.getChildren():
```

```
            if isinstance(child, KobraV2Parser.InstruccionContext):
                self.visit(child)

        # 4. Actualización (al final del while)
        update_stmt = ctx.asignacion_for(1)
        var_update = update_stmt.ID().getText()
        val_update = self.visit(update_stmt.expresion())
        # Usamos add_line normal, ya tiene el indent_level aumentado
        self.add_line(f"{var_update} = {val_update}")

        self.indent_level -= 1
        return None

    def visitBloque(self, ctx):
        self.indent_level += 1
        self.visitChildren(ctx)
        self.indent_level -= 1
        return None

    # --- EXPRESIONES ---
    # Retornan strings con el código de la expresión

    def visitParentesis(self, ctx):
        return f"({self.visit(ctx.expresion())})"

    def visitOpAditiva(self, ctx):
        left = self.visit(ctx.expresion(0))
        right = self.visit(ctx.expresion(1))
        op = ctx.op.text
        return f"{left} {op} {right}"

    def visitOpMultiplicativa(self, ctx):
        left = self.visit(ctx.expresion(0))
        right = self.visit(ctx.expresion(1))
        op = ctx.op.text

        if op == '/':
```

```
            return f"_kobra_div({left}, {right})"

        return f"{left} {op} {right}"

    def visitOpRelacional(self, ctx):
        left = self.visit(ctx.expresion(0))
        right = self.visit(ctx.expresion(1))
        op = ctx.op.text
        return f"{left} {op} {right}"

    def visitOpIgualdad(self, ctx):
        left = self.visit(ctx.expresion(0))
        right = self.visit(ctx.expresion(1))
        op = ctx.op.text
        # En Kobra '==' es igual a Python, pero si fuera distinto aquí
se cambia
        return f"{left} {op} {right}"

    def visitOpLogica(self, ctx):
        left = self.visit(ctx.expresion(0))
        right = self.visit(ctx.expresion(1))
        op = ctx.op.text
        # Traducir Y/O a and/or
        py_op = "and" if op == 'Y' else "or"
        return f"{left} {py_op} {right}"

    def visitVariable(self, ctx):
        return ctx.ID().getText()

    def visitEntero(self, ctx):
        return ctx.INT().getText()

    def visitFlotante(self, ctx):
        return ctx.FLOAT().getText()

    def visitCadena(self, ctx):
        return ctx.STRING().getText()
```

```
    def visitBooleano(self, ctx):
        val = ctx.BOOL_LIT().getText()
        return "True" if val == "VERDADERO" else "False"
```

main.py

```
import sys
from antlr4 import *
from KobraV2Lexer import KobraV2Lexer
from KobraV2Parser import KobraV2Parser
from KobraCompiler import KobraToPythonVisitor


def main():
    if len(sys.argv) < 2:
        print("Uso: python main.py <archivo_input.txt>")
        return

    input_file = sys.argv[1]
    output_file = input_file.replace('.txt', '.py')

    # 1. Leer input
    input_stream = FileStream(input_file, encoding='utf-8')

    # 2. Lexer
    lexer = KobraV2Lexer(input_stream)
    stream = CommonTokenStream(lexer)

    stream.fill()
    print("\n--- Tokens identificados ---")
    for token in stream.tokens:
        print(token.text)
    print("\n")

    # 3. Parser
    parser = KobraV2Parser(stream)
```

```
    tree = parser.programa()  # 'programa' es la regla raiz

    # Verificar errores de sintaxis
    if parser.getNumberOfSyntaxErrors() > 0:
        print("Errores de sintaxis encontrados. No se generó código.")
        return

    # 4. Transpilación (Visitor)
    visitor = KobraToPythonVisitor()
    python_code = visitor.visit(tree)

    # 5. Guardar Output
    with open(output_file, 'w', encoding='utf-8') as f:
        f.write(python_code)

    print(f"Transpilación exitosa! Archivo generado: {output_file}")


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

6. Troubleshooting

Caso: `Errores de sintaxis encontrados. No se generó código.`

Causa: El programa encontró errores durante el proceso de construcción de expresiones en el análisis sintáctico.

Solución: El programa enviara una o múltiples líneas indicando en que línea del programa ocurrió el error de sintáctico para que pueda ser corregido por el usuario:

`line <línea>:<carácter> missing ‘<token>’ at ‘<token>’`

Caso: `FileNotFoundError: [Errno 2] No such file or directory: '<archivo>'`

Causa: El programa no encontró el archivo especificado en el comando de ejecución.

Solución: Asegurarse de escribir el nombre completo del archivo junto con la terminación `.txt` cuando se intente ejecutarse el archivo main.py

```
python main.py <Nombre de archivo>.txt
```

Caso: `UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd2 in position 16: invalid continuation byte`

`decoding with 'utf-8' codec failed`

Causa: El programa esta intentando leer un archivo que no cuenta con un formato de codificación utf-8 por lo cual es incompatible

Solución: Asegurarse de utilizar archivos con la terminación `.txt`

7. Preguntas frecuentes

- *¿Por qué se requiere de la función `kobra_div` para la funcionalidad del transpilador?*

R: Debido a que Python asigna variables de forma dinámica, lo que significa que una misma variable pueden cambiar completamente su tipo de dato durante las distintas partes de un programa, al realizar una división entre dos valores enteros Python automáticamente convertiría el resultado en un valor con punto decimal de tipo flotante. Para esto la función `kobra_div` se encarga de verificar si ambos valores son de tipo entero y entonces realiza una división sin punto decimal con el operador `//`.

- *¿Por qué se decidió permitir solamente entradas del programa en archivos `.txt`?*

R: La razón fue debido a que esto permite una mayor facilidad al traducir varias líneas de código y porque la mayoría de los sistemas modernos cuentan con un editor de archivos `.txt` integrado.

- *¿Por qué solamente se incluyeron las sentencias que incluyeron?*

R: Esto se hizo para delimitar el desarrollo de este proyecto a una escala la cual estuvimos seguros de que podríamos desarrollar en tiempo y forma, pero sin la necesidad de excluir funciones básicas como los ciclos condicionales, sentencias condicionales y funciones para la visualización de datos como `print`.

- *¿Por qué decidieron agregar a la gramática diferencias con el lenguaje Python, tales como el punto y coma para indicar el fin de una sentencia?*

R: Esta decisión se tomo para simplificar tanto la escritura como el proceso de análisis del código escrito de nuestro lenguaje.

- *¿Por qué se utilizó un visitor en lugar de un listener?*

R: Debido a que esta clase nos permite tener un más preciso control de la forma en que se recorren los nodos dentro del árbol generado mediante el analizador sintáctico.

- *¿Por qué se separaron las expresiones en diferentes tipos (Aditivas, multiplicativas, etc.)?*

R: Porque de esta manera se está implementando la precedencia de operadores dentro de la misma expresión, con el orden en el que se encuentran ordenadas definiendo cuales operaciones tienen mayor prioridad.

- *¿Por qué la instrucción equivalente del `for` (PARA) hace uso de una sentencia `while`?*

R: Debido a la manera en que el lenguaje Python hace uso de la sentencia `for` para recorrer una estructura de datos, decidimos implementar el equivalente de la sentencia `for` la cual es un ciclo `while` que hace uso de un contador como la condición para concluir las repeticiones de un bloque de código.