	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: Compiladores

Grupo: 04

No de Práctica(s): Intérprete

Integrante(s): 318307921

318035327

318051363

318045351

No. de lista o brigada: Equipo 11

Semestre: 2024-1

Fecha de entrega: 04 de Diciembre de 2023

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

El objetivo principal de este documento es desarrollar la implementación de un intérprete por medio de las etapas de construcción esenciales: un analizador léxico, un analizador sintáctico y un analizador semántico. Para ello, se hará énfasis en el concepto de analizador semántico y del linker-ejecutador que son los elementos a incorporar al analizador léxico y sintáctico previamente desarrollados.

La implementación del intérprete se trabajó en el lenguaje Python con una programación modular compuesta de métodos y funciones y aplicando elementos de paradigma orientado a objetivos, por lo cual se mencionan algunas bibliotecas importantes que permitieron el desarrollo del proyecto.

También, se incluyen fragmentos relevantes acerca del funcionamiento del proyecto. Este análisis se presenta para la gramática propuesta posteriormente.

1.1. Hipótesis

El intérprete generado deberá retomar los resultados de los analizadores generadores anteriormente para realizar su análisis semántico para determinar las variables y etiquetas presentan un uso correcto y el intérprete deberá ejecutar las instrucciones analizadas y mostrará en pantalla una frase que indique lo que se está ejecutando para saber que se ha implementado exitosamente.

2. Desarrollo

2.1. Revisión Documental

2.1.1. ¿Qué es un analizador semántico?

La fase de análisis semántico de un procesador de lenguaje es aquélla que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación.

La semántica de un lenguaje de programación es el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente válida. Finalmente,

el análisis semántico de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico. [1]

2.1.2. ¿Qué es el linker-ejecutador?

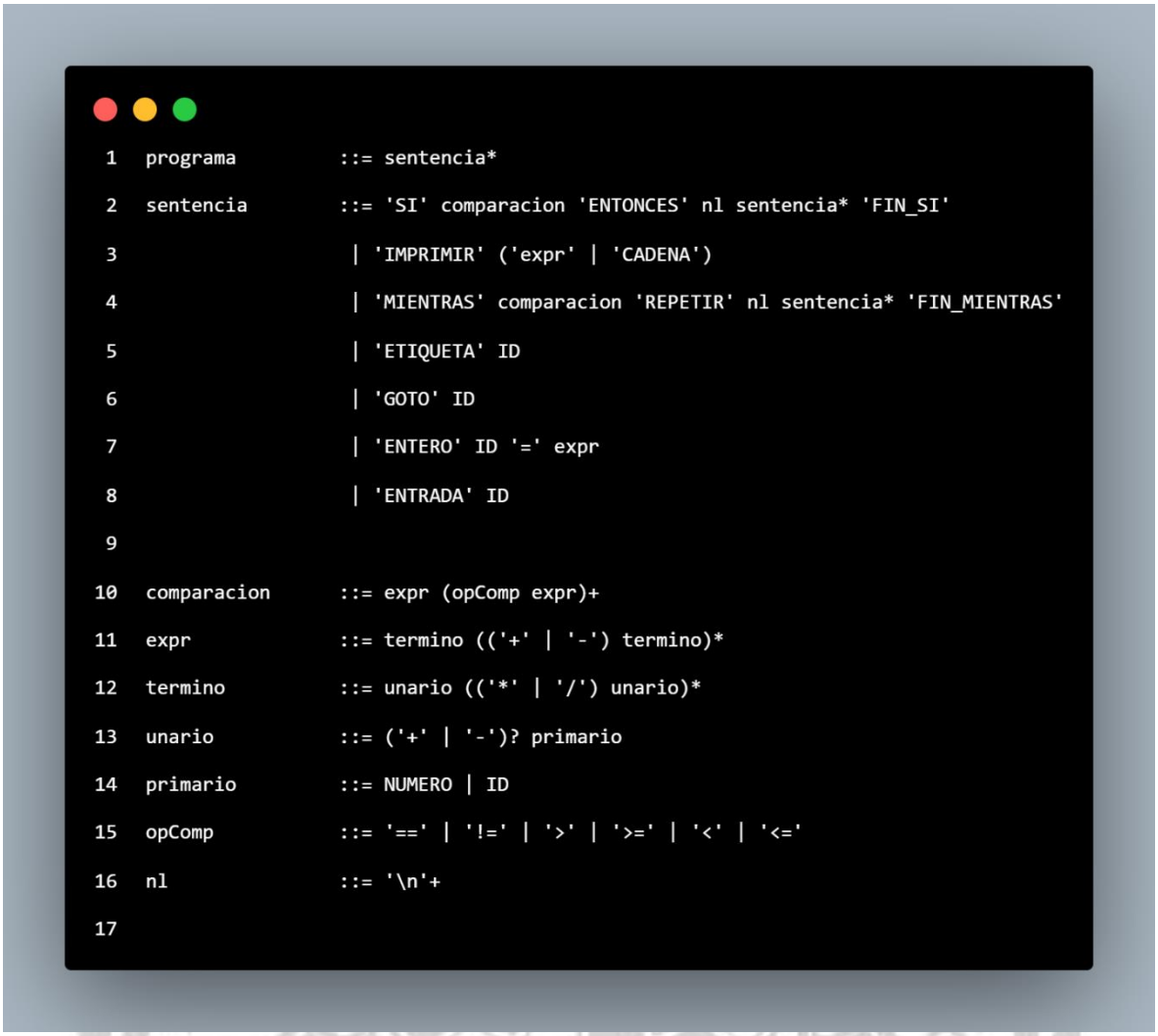
En el contexto de los intérpretes, a veces se puede hacer referencia a la "vinculación dinámica" o "enlace dinámico". Esto ocurre cuando, durante la ejecución, ciertas bibliotecas o módulos se cargan dinámicamente en tiempo de ejecución para extender la funcionalidad del programa. Aunque este concepto se asemeja a la idea de enlace, no se realiza de la misma manera que en un sistema compilado con un linker.

En lugar de vincular todas las bibliotecas de antemano, como en el enlace estático en compiladores, un intérprete puede cargar o vincular bibliotecas según sea necesario durante la ejecución. Esto proporciona flexibilidad y eficiencia en términos de recursos.

Es importante destacar que el término "linker" en el contexto de intérpretes puede ser utilizado de manera menos convencional y específica para referirse a la gestión dinámica de módulos o extensiones durante la ejecución. La implementación específica puede variar según el entorno y el lenguaje de programación utilizado. [2]

2.1.3. Gramática Utilizada

A continuación, se presenta una representación de la gramática para un lenguaje de programación básico, utilizando la notación Backus-Naur Form (BNF):



```
programa        ::= sentencia*
sentencia       ::= 'SI' comparacion 'ENTONCES' nl sentencia* 'FIN_SI'
                  | 'IMPRIMIR' ('expr' | 'CADENA')
                  | 'MIENTRAS' comparacion 'REPETIR' nl sentencia* 'FIN_MIENTRAS'
                  | 'ETIQUETA' ID
                  | 'GOTO' ID
                  | 'ENTERO' ID '=' expr
                  | 'ENTRADA' ID

comparacion     ::= expr (opComp expr)+
expr            ::= termino (('+' | '-') termino)*
termino         ::= unario (('*' | '/') unario)*
unario          ::= ('+' | '-')? primario
primario        ::= NUMERO | ID
opComp          ::= '==' | '!=' | '>' | '>=' | '<' | '<='
nl              ::= '\n'+
```

2.2. Bibliotecas Utilizadas

2.2.1. Biblioteca enum

La clase Enum de la biblioteca enum se utiliza para definir enumeraciones. Estas enumeraciones son una forma de crear un conjunto de nombres constantes que pueden ser más significativos y legibles que los valores numéricos.

TipoToken Enum:

En nuestro código, TipoToken es una enumeración que representa los diferentes tipos de tokens que pueden ser identificados durante el análisis léxico. Cada elemento de esta enumeración tiene un nombre (por ejemplo, FIN_LINEA, SALTO_LINEA, NUMERO, etc.) y un valor asociado. Los valores están asignados automáticamente por la biblioteca enum, pero se puede acceder a ellos utilizando el atributo value.

Algunos ejemplos de elementos en la enumeración TipoToken y sus valores asociados son:

- FIN_LINEA con valor -1
- SALTO_LINEA con valor 0
- NUMERO con valor 1
- ID con valor 2
- ...

Esta enumeración proporciona una forma clara y semánticamente significativa de referirse a los diferentes tipos de tokens en lugar de usar valores numéricos directamente. Por ejemplo, en lugar de comparar si el tipo de token es igual a 1 para verificar si es un número, puedes usar TipoToken.NUMERO. Esto hace que el código sea más legible y menos propenso a errores.

En resumen, la biblioteca enum y la enumeración TipoToken se utilizan para mejorar la claridad y mantenibilidad del análisis léxico al proporcionar nombres significativos para los diferentes tipos de tokens.

2.2.2. Biblioteca sys

La biblioteca sys en Python proporciona acceso a algunas variables utilizadas o mantenidas por el intérprete de Python y funciones que interactúan fuertemente con el intérprete. En nuestro código, se utiliza la función sys.exit(mensaje) para salir del programa si ocurre un error.

```
import sys
sys.exit(mensaje):
```

- Esta función se utiliza para salir del programa de manera inmediata.
- "mensaje es opcional y se imprime en la salida estándar (por lo general, la consola) antes de finalizar el programa.
- Se utiliza comúnmente para indicar un error crítico y detener la ejecución del programa.

Ejemplo de uso en nuestro código:

```
def abortar(self, mensaje):
    sys.exit("Error: " + mensaje)
```

En este caso, si la función abortar es llamada, el programa se cerrará inmediatamente con un mensaje de error indicando la razón de la terminación.

2.3. Fragmentos importantes del analizador léxico

Este código implementa un analizador léxico para nuestro lenguaje de programación Quasar. Aquí hay una descripción de las principales funciones y clases:

Clase Lexico:

1. **Constructor (__init__):**
   - Inicializa el objeto Lexico con el código fuente proporcionado.
   - Añade un carácter de nueva línea (\n) al final del código fuente para facilitar el análisis del último token o sentencia.
   - Llama al método siguiente () para inicializar el primer carácter.

2. **Método siguiente:**
   - Mueve la posición actual al siguiente carácter en el código fuente.
   - Actualiza el carácter actual con el nuevo carácter en la posición actual.
3. **Método asomar:**
   - Devuelve el siguiente carácter sin cambiar la posición actual.
4. **Método abortar:**
   - Finaliza el programa y muestra un mensaje de error si se encuentra un token inválido.
5. **Método saltarEspacios:**
   - Avanza la posición actual mientras se encuentren espacios en blanco, tabuladores y otros caracteres de espacio, pero no avanza en caso de encontrar una nueva línea.
6. **Método saltarComentarios:**
   - Salta los comentarios en el código. Si encuentra el carácter #, se mueve hacia adelante hasta encontrar un salto de línea (\n).
7. **Método getToken:**
   - Devuelve el siguiente token del código fuente.

- Llama a saltarEspacios y saltarComentarios para omitir espacios y comentarios antes de identificar el próximo token.
- Identifica los tokens basándose en los caracteres actuales y los siguientes.
- Actualiza la posición y el carácter actuales después de identificar un token.

Clase Token:

1. **Constructor (__init__):**
   - Inicializa un objeto Token con un lexema y un tipo de token (TipoToken).
   - Método estático revisarSiKeywords:
   - Recibe un lexema y verifica si es una palabra clave (keyword) basada en la enumeración TipoToken.
   - Devuelve el tipo de token correspondiente si el lexema es una palabra clave; de lo contrario, devuelve None.

2. **Enumeración TipoToken:**
   - Enumera todos los tipos de tokens posibles, como fin de línea (FIN_LINEA), salto de línea (SALTO_LINEA), números (NUMERO), identificadores (ID), cadenas (CADENA), y palabras clave específicas del lenguaje ficticio.
   - También enumera operadores y sus combinaciones.

```python
class Token:
    def __init__(self, lexema, token):
        self.lexema = lexema
        self.token = token  # TipoToken ENUM

    @staticmethod
    def revisarSiKeywords(lexema):
        # Usar la enumeracion: TipoToken.name(nombre); TipoToken.value(numeros)
        for tipo in TipoToken:
            if tipo.name == lexema and tipo.value > 100 and tipo.value < 200:
                return tipo
        return None


class TipoToken(enum.Enum):
    # Escribir todos los tokens
    FIN_LINEA = -1  # End of file #\n
    SALTO_LINEA = 0  # \n
    NUMERO = 1  # si
    ID = 2
    CADENA = 3  # si
    # Keywords 100>, pero <200
    ETIQUETA = 101
    GOTO = 102
    IMPRIMIR = 103
    ENTRADA = 104
    ENTERO = 105
    SI = 106
    ENTONCES = 107
    FINSI = 108
    MIENTRAS = 109
    REPETIR = 110
    FINMIENTRAS = 111
    # Operadores
    IGUAL = 201  # = 2
    MAS = 202  # +
    MENOS = 203  # -
    ASTERISCO = 204  # *
    DIAGONAL = 205  # /
    IGUAL_IGUAL = 206  # == 2
    NO_IGUAL = 207  # != 2
    MENOR = 208  # < 2
    MENOR_IGUAL = 209  # <= 2
    MAYOR = 210  # > 2
    MAYOR_IGUAL = 211  # >= 2

```

2.4. Fragmentos Importantes del analizado sintáctico

Este código implementa un analizador sintáctico para nuestro lenguaje de programación Quasar. El analizador sintáctico se encarga de analizar la estructura gramatical de un programa fuente, que se le proporciona a través de un ***analizador léxico (lexico)***. La clase Sintatico contiene varios métodos que implementan las reglas de producción del lenguaje. Aquí está un resumen de las principales funcionalidades:

1. Método programa:
    - Inicia el análisis sintáctico del programa fuente.
    - Itera a través de las sentencias del programa hasta encontrar un token de fin de línea (FIN_LINEA).

- Después de analizar el programa, se llama a semantico.revisarLabelsGoto() para verificar las etiquetas utilizadas en instrucciones GOTO.

2. Método sentencia:
   - Implementa las reglas de producción para diversas instrucciones del lenguaje ficticio.
   - Las instrucciones incluyen: SI, IMPRIMIR, MIENTRAS, ETIQUETA, GOTO, ENTERO, y ENTRADA.
   - Para cada instrucción, se realiza un análisis y se ejecutan las acciones semánticas y de ejecución correspondientes utilizando instancias de las clases Semantico y Ejecutor.
3. Método comparacion:
   - Implementa la regla de producción para comparaciones, como las usadas en instrucciones SI y MIENTRAS.
   - Utiliza recursión para manejar operaciones de comparación con múltiples términos.
4. Método expr, termino, unario, primario:
   - Implementan las reglas de producción para expresiones aritméticas.
   - Manejan operadores aritméticos como suma, resta, multiplicación y división.
   - expr es la regla de producción más alta y llama a las otras reglas de producción de manera jerárquica.
5. Método opComp:
   - Implementa la regla de producción para operadores de comparación como ==, !=, >, >=, <, y <=.
6. Método nl:
   - Implementa la regla de producción para nuevas líneas (\n).
   - Se asegura de que haya al menos una nueva línea y consume las líneas adicionales.

```python
def sentencia(self):
        # ‘SI’ comparación ‘ENTONCES’ nl (sentencia)* ‘FIN_SI’
        if self.revisarToken(TipoToken.SI):  # revisarToken (SI/MIENTRAS) y regresa true
            # print("SI ", end='')
            self.siguienteToken()
            self.comparacion()

            self.match(TipoToken.ENTONCES)  # Si es pasa al siguiente; si no es, error
            self.nl()

            # (sentencia)*
            while not self.revisarToken(TipoToken.FINSI):
                self.sentencia()

            self.match(TipoToken.FINSI)

        # ‘IMPRIMIR’  (expr | CADENA) == 'IMPRIMIR' expr | 'IMPRIMIR' CADENA
        elif self.revisarToken(TipoToken.IMPRIMIR):
            # print("IMPRMIR ", end='')
            self.siguienteToken()
            if self.revisarToken(TipoToken.CADENA):
                mensaje1 = self.tokenActual.lexema[1:-1]  # Eliminar las comillas
                self.siguienteToken()
            else:
                self.expr()
            ejecutor.imprimir({"mensaje": mensaje1})

        # ‘MIENTRAS’ comparación ‘REPETIR’ nl (sentencia)* ‘FIN_MIENTRAS’
        elif self.revisarToken(TipoToken.MIENTRAS):
            # print("MIENTRAS ", end='')
            self.siguienteToken()
            self.comparacion()
            self.match(TipoToken.REPETIR)
            self.nl()
            while not self.revisarToken(TipoToken.FINMIENTRAS):  # sentencia*
                self.sentencia()
            self.match(TipoToken.FINMIENTRAS)

        # 'LABEL' ID (Etiqueta)
        elif self.revisarToken(TipoToken.ETIQUETA):
            # print("ETIQUETA ", end='')
            self.siguienteToken()
            # Checar que la etiqueta no exista ya
            semantico.revisarLabelDeclarada(self.tokenActual.lexema)
            semantico.agregarLabelDeclarada(self.tokenActual.lexema)
            self.match(TipoToken.ID)

            # 'GOTO' ID (salto)
        elif self.revisarToken(TipoToken.GOTO):
            # print("GOTO ", end='')
            self.siguienteToken()
            semantico.agregarLabelGoto(self.tokenActual.lexema)
            self.match(TipoToken.ID)

        # 'ENTERO' ID '=' expr == ['ENTERO' ID IGUAL expr]
        elif self.revisarToken(TipoToken.ENTERO):
            # print("ENTERO ", end='')
            self.siguienteToken()
            nombre_variable1 = self.tokenActual.lexema
            semantico.agregarVariable(self.tokenActual.lexema)

            self.match(TipoToken.ID)
            self.match(TipoToken.IGUAL)
            valor1 = self.expr()
            ejecutor.declarar_variable({"nombre": nombre_variable1, "valor": valor1})
        # 'INPUT' ID
        elif self.revisarToken(TipoToken.ENTRADA):
            # print("ENTRADA ", end='')
            self.siguienteToken()
            nombre_variable = self.tokenActual.lexema
            semantico.agregarVariable(self.tokenActual.lexema)
            ejecutor.entrada_usuario({"nombre": nombre_variable})

            self.match(TipoToken.ID)
        else:
            self.abortar(" 1 Sentencia no válida en " + self.tokenActual.lexema + "(" + self.tokenActual.token.name + ")")

        # Newline final
        self.nl()
```

2.5. Fragmentos importantes del analizador semántico

Este código define una clase llamada Semantico, que se utiliza para realizar verificaciones semánticas básicas de nuestro lenguaje de programación Quasar. Las verificaciones semánticas se centran en la correctitud y coherencia de las variables y etiquetas utilizadas en el código.

A continuación, se describen las funciones y su propósito en la clase:

1. **Constructor (__init__):**
   Inicializa tres conjuntos vacíos: variables para almacenar variables, labelsDeclaradas para almacenar etiquetas declaradas, y labelsGoto para almacenar etiquetas a las que se ha saltado con la instrucción GOTO.

2. **Método abortar:**
   Toma un mensaje como argumento y finaliza la ejecución del programa mostrando un mensaje de error que indica un problema semántico.

3. **Método revisarVariable:**
   Toma el nombre de una variable como argumento y verifica si la variable ha sido declarada. Si no ha sido declarada, aborta el programa con un mensaje de error.

4. **Método agregarVariable:**
   Toma el nombre de una variable como argumento y la agrega al conjunto de variables.

5. **Método revisarLabelDeclarada:**
   Toma el nombre de una etiqueta como argumento y verifica si la etiqueta ya ha sido declarada. Si ya está declarada, aborta el programa con un mensaje de error.

6. **Método agregarLabelDeclarada:**
   Toma el nombre de una etiqueta como argumento y la agrega al conjunto de etiquetas declaradas.

7. **Método revisarLabelsGoto:**
   Verifica si todas las etiquetas a las que se ha saltado (labelsGoto) han sido declaradas previamente (labelsDeclaradas). Si alguna etiqueta no está declarada, aborta el programa con un mensaje de error.

8. **Método agregarLabelGoto:**

   Toma el nombre de una etiqueta como argumento y la agrega al conjunto de etiquetas a las que se ha saltado (labelsGoto).

```python
import sys
class Semantico:
    def __init__(self):
        self.variables = set()  # Variables
        self.labelsDeclaradas = set()  # Labels
        self.labelsGoto = set()  # Labels a las que se a saltado (GOTO)

    def abortar(self, mensaje):
        sys.exit("Error: " + mensaje)

    def revisarVariable(self, variable):
        if variable not in self.variables:
            self.abortar("La variable no ha sido declarada: " + variable)

    def agregarVariable(self, variable):
        if variable not in self.variables:
            self.variables.add(variable)

    def revisarLabelDeclarada(self, etiqueta):
        if etiqueta in self.labelsDeclaradas:
            self.abortar("Ese Label (Etiqueta) ya existe: " + etiqueta)

    def agregarLabelDeclarada(self, etiqueta):
        self.labelsDeclaradas.add(etiqueta)  # Agregando las labels declaradas

    def revisarLabelsGoto(self):
        for etiqueta in self.labelsGoto:
            if etiqueta not in self.labelsDeclaradas:
                self.abortar("Se intenta saltar a una etiqueta que no esta declarada con nombre: " + etiqueta)

    def agregarLabelGoto(self, etiqueta):
        self.labelsGoto.add(etiqueta)  # Agregando el salto
```

2.6. Fragmentos importantes del linker-ejecutador

La clase Ejecutor tiene un constructor __init__ que inicializa un diccionario llamado variables para almacenar las variables del programa. La ejecución del programa se realiza mediante el método ejecutar_programa, que itera a través de las instrucciones y llama a métodos específicos según el tipo de instrucción.

Las instrucciones pueden ser de tres tipos: "DECLARACION", "ENTRADA" e "IMPRIMIR".

- La función declarar_variable se encarga de procesar las instrucciones de declaración, asignando un valor a una variable y mostrando un mensaje de depuración.
- La función entrada_usuario maneja las instrucciones de entrada, solicitando al usuario que ingrese un valor para una variable específica, convirtiéndolo a entero y almacenándolo en el diccionario de variables. También imprime un mensaje de depuración.
- La función imprimir simplemente imprime el mensaje asociado a una instrucción de tipo "IMPRIMIR".

```python
class Ejecutor:
    def __init__(self):
        self.variables = {}  # Almacena las variables

    def ejecutar_programa(self, instrucciones):
        for instruccion in instrucciones:
            if instruccion["tipo"] == "DECLARACION":
                self.declarar_variable(instruccion)
            elif instruccion["tipo"] == "ENTRADA":
                self.entrada_usuario(instruccion)
            elif instruccion["tipo"] == "IMPRIMIR":
                self.imprimir(instruccion)
    # Implementado Exitosamente
    def declarar_variable(self, instruccion):
        nombre_variable = instruccion["nombre"]
        valor = instruccion["valor"]
        self.variables[nombre_variable] = valor #declaración de la variable

        print(f"[Depuración] Variable declarada {nombre_variable} con valor {self.variables[nombre_variable]}")
    #Implementado Exitosamente
    def entrada_usuario(self, instruccion):
        nombre_variable = instruccion["nombre"]
        entrada = input(f"Ingrese el valor de {nombre_variable}: ")
        self.variables[nombre_variable] = int(entrada)
        print(f"[Depuración] El valor de {nombre_variable} es {self.variables[nombre_variable]}")
    # Función implementada correctamente
    def imprimir(self, instruccion):
        mensaje = instruccion["mensaje"]
        print(mensaje)


```

2.7. Archivo de entrada

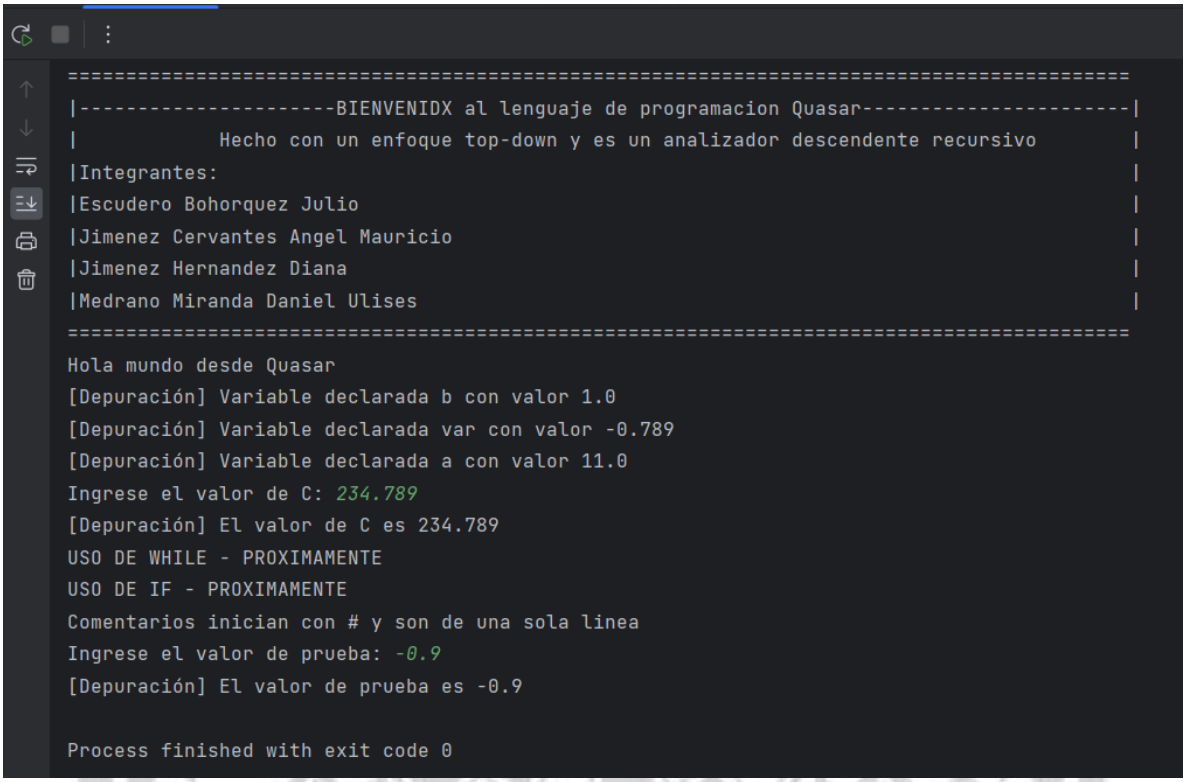
Este archivo es una cadena valía para nuestro lenguaje de programación

```
IMPRIMIR " ================================================================================================ "

IMPRIMIR " |----------------------BIENVENIDX al lenguaje de programacion Quasar-----------------------| "

IMPRIMIR " |            Hecho con un enfoque top-down y es un analizador descendente recursivo        | "

IMPRIMIR " |Integrantes:                                                                               | "

IMPRIMIR " |Escudero Bohorquez Julio                                                                   | "

IMPRIMIR " |Jimenez Cervantes Angel Mauricio                                                           | "

IMPRIMIR " |Jimenez Hernandez Diana                                                                    | "

IMPRIMIR " |Medrano Miranda Daniel Ulises                                                              | "

IMPRIMIR " ================================================================================================  "

IMPRIMIR " Hola mundo desde Quasar "
ENTERO b = 1
ENTERO var=-0.789
#Las palabras reservadas son ENTERO que su equivalente en ejemplo  en C sería la declaración de variables int
ENTERO a = 11


#Declaración de variables para entrada del usuario
ENTRADA C


#Ciclo while
MIENTRAS a != 0 REPETIR
IMPRIMIR " USO DE WHILE - PROXIMAMENTE "
FINMIENTRAS

#Condicional IF
    SI a <= 10 ENTONCES
#Mandar a imprimir en pantalla
    IMPRIMIR " USO DE IF - PROXIMAMENTE "
    FINSI


#Mandar a imprimir en pantalla
IMPRIMIR " Comentarios inician con # y son de una sola linea "


#Tipo Rollback en una base de datos

GOTO initio


#Aqui aplica como un checkpoint como en una Base de Datos
ETIQUETA initio

#Comentarios
#hola

ENTRADA prueba
```

2.8. SALIDA FINAL

Las impresiones que dicen Depuración son para verificar que la variable se declarará correctamente

Las estructuras de control IF y WHILE se agregará próximamente.



3. Conclusión

En conclusión, se ha logrado desarrollar un intérprete básico que ejecuta las instrucciones analizadas léxica, sintáctica y semánticamente, pudiendo incorporar dichos elementos satisfactoriamente. De esta forma, fue posible implementar cada una de las etapas de análisis exitosamente de una gramática propuesta generando un lenguaje de programación muy básico que puede ser interpretado.

Este interprete es capaz de distinguir entre tres tipos de instrucciones, declaraciones, entradas de usuario e impresiones, sin embargo, se espera que posteriormente puedan ser interpretadas declaraciones if y while.

El desarrollo de este proyecto permite un aprendizaje más profundo sobre las etapas de análisis que comparten el compilador, intérprete y traductor.

4. Repositorio de Github

https://github.com/Dumm312/Equipo11_Compiladores_Grupo4.git

5. Referencias

[1] “Análisis Semántico en Procesadores de Lenguaje”. Universidad de Oviedo. Accedido el 4 de diciembre de 2023. [En línea]. Disponible: https://reflection.uniovi.es/ortin/publications/semantico.pdf

[2] “¿Cuáles son los tipos de interpretación?”, Lingua serve. Accedido el 4 de diciembre de 2023. [En línea]. Disponible: https://blog.linguaserve.com/cuales-son-los-tipos-de-interpretaci%C3%B3n