INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Computo

PROYECTO

Generador de figuras

Asignatura:
Compiladores

Presenta:
Bernal Ramírez Brian Ricardo
Escalona Zuñiga Juan Carlos

Profesor:
Tecla Parra Roberto

25 / Junio /2025

Contenido

Introducción.

En el ámbito de la informática, un compilador es una herramienta esencial que transforma el código fuente escrito en un lenguaje de programación de alto nivel en un lenguaje de máquina entendible por el hardware de una computadora. Los compiladores no solo traducen el código, sino que también optimizan el rendimiento del programa y verifican la sintaxis y la semántica del código fuente para detectar errores. Los lenguajes de programación como C, Java, y Python dependen de compiladores (o intérpretes en el caso de Python) para ejecutar programas.

Módulos del compilador.

Funcion.java

Explicación

El módulo Funcion.java define una interfaz en Java llamada Funcion. Esta interfaz especifica que cualquier clase que la implemente debe proporcionar una implementación del método ejecutar. Este método acepta dos parámetros: un objeto genérico Object y una lista de parámetros ArrayList.

Explicación de su unión con otros módulos

La interfaz Funcion se utiliza en otros módulos como MaquinaDePila.java, donde diferentes implementaciones de funciones específicas (como Girar, Avanzar, CambiarColor, SubirPincel, BajarPincel) son definidas e invocadas en el contexto de una máquina de pila para ejecutar comandos en un entorno gráfico.

Este módulo es crucial ya que define la estructura que deben seguir todas las funciones ejecutables dentro del sistema, proporcionando una forma estándar de implementar y llamar a funciones dentro de la máquina de pila. La implementación de esta interfaz en diferentes clases permite la ejecución de comandos específicos necesarios para la funcionalidad de dibujo y control dentro del programa.

MaquinaDePila.java

Explicación

El módulo MaquinaDePila.java implementa una máquina de pila que ejecuta instrucciones en una estructura de pila. Esta clase gestiona la memoria, la pila de ejecución, y los marcos de funciones. Proporciona métodos para agregar y ejecutar operaciones, manejar variables, realizar operaciones aritméticas y lógicas, y ejecutar estructuras de control como bucles y condicionales.

Explicación de su unión con otros módulos

- Funcion.java: Implementa varias funciones (Girar, Avanzar, etc.) que son ejecutadas por la máquina de pila.
- TablaDeSimbolos.java: Utiliza una tabla de símbolos para gestionar las variables y funciones definidas.
- Configuracion.java y Linea.java: Estas clases se usan para definir la configuración de dibujo y los trazos que se deben ejecutar.

Partes más relevantes del código

Método agregarOperacion:

```
public int agregarOperacion(String nombre){
    int posicion = memoria.size();
    try{
        memoria.add(this.getClass().getDeclaredMethod(nombre, null));
        return posicion;
    } catch(Exception e) {
        System.out.println("Error al agregar operación " + nombre + ". ");
    }
    return -1;
}
```

- Agrega una operación (método) a la memoria.
- Usa reflexión para encontrar y agregar el método por nombre. **Método**

SUM:

```
public void ejecutar() {
    stop = false;
    while(contadorDePrograma < memoria.size())
        ejecutarInstruccion(contadorDePrograma);
}
```

Realiza la suma de los dos elementos superiores de la pila y empuja el resultado de vuelta a la pila.

Método ejecutar:

```
public void ejecutar() {
    stop = false;
    while(contadorDePrograma < memoria.size())
        ejecutarInstruccion(contadorDePrograma);
}
```

Ejecuta las instrucciones almacenadas en lamemoria secuencialmente.

Método ejecutarInstruccion:

```
public void ejecutarInstruccion(int indice) {
    try {
        Object objetoLeido = memoria.get(indice);
        if(objetoLeido instanceof Method){
            Method metodo = (Method)objetoLeido;
            metodo.invoke(this, null);
        }
        if(objetoLeido instanceof Funcion){
            ArrayList parametros = new ArrayList();
            Funcion funcion = (Funcion)objetoLeido;
            contadorDePrograma++;
            while(memoria.get(contadorDePrograma) != null) {
                if(memoria.get(contadorDePrograma) instanceof String){
                    if(((String)(memoria.get(contadorDePrograma))).equals("Limite")){
                        Object parametro = pila.pop();
                        parametros.add(parametro);
                        contadorDePrograma++;
                    }
                } else {
                    ejecutarInstruccion(contadorDePrograma);
                }
            }
            funcion.ejecutar(configuracionActual, parametros);
        }
        contadorDePrograma++;
    } catch(Exception e) {}
}
```

- Determina el tipo de instrucción (método o función) y la ejecuta.
- Usa reflexión para invocar métodos.
- Maneja la ejecución de funciones con parámetros.

Este módulo es fundamental para la ejecución de instrucciones en el programa. Proporciona la infraestructura para manejar operaciones, variables, y funciones, permitiendo la ejecución dinámica de comandos en un entorno de dibujo.

Marco.java

Explicación

El módulo Marco.java define la clase Marco, que es utilizada para representar un marco de función en la máquina de pila. Un marco de función almacena los parámetros de la función, el punto de retorno y el nombre de la función, permitiendo la correcta gestión y ejecución de funciones dentro de la máquina de pila.

Explicación de su unión con otros módulos

Marco.java se utiliza dentro del módulo MaquinaDePila.java para gestionar la ejecución de funciones. Cada vez que se invoca una función, se crea un nuevo marco que almacena el contexto de esa llamada, incluyendo los parámetros y el punto de retorno. Este marco se agrega a una pila de marcos (pilaDeMarcos) que

permite a la máquinade pila manejarllamadasy retornos defuncionesdemanera anidada.

Partes más relevantes del código

Métodos de gestión de parámetros:

```
public void agregarParametro(Object parametro) {
    parametros.add(parametro);
}

public Object getParametro(int i) {
    return parametros.get(i);
}

public void setParametros(ArrayList parametros) {
    this.parametros = parametros;
}
```

- agregarParametro: Agrega un parámetro a la lista de parámetros del marco.
- getParametro: Obtiene el parámetro en la posición i.
- setParametros: Establece la lista completa de parámetros. **Métodos**

de gestión del punto de retornoy el nombre:

```
public int getRetorno() {
    return retorno;
}
public void setRetorno(int retorno) {
    this.retorno = retorno;
}
public String getNombre() {
    return nombre;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
```

- getRetorno y setRetorno: Obtienen y establecen el punto de retorno.
- getNombre y setNombre: Obtienen y establecen el nombre de la función.

Este módulo es esencial para la ejecución estructurada de funciones dentro de la máquina de pila, permitiendo que se mantenga el contexto adecuado para cada llamada de función y se manejen correctamente los retornos.

P2.y

Explicación

El archivo P2.y es un archivo de especificación YACC que define la gramática y las acciones semánticas para el lenguaje Logos. Este archivo es utilizado por el compilador YACC para generar un parser que puede analizar y ejecutar scripts

escritos en el lenguaje Logos. Define reglas de producción para expresiones, declaraciones y estructuras de control, así como las acciones que deben tomarse cuando se reconocen estas estructuras.

Explicación de su unión con otros módulos

P2.y genera el parser que se implementa en el módulo Parser.java. Este parser utiliza la máquina de pila (MaquinaDePila.java) para ejecutar las instrucciones reconocidas. También interactúa con la tabla de símbolos (TablaDeSimbolos.java) para manejar variables y funciones, y con la configuración (Configuracion.java) para ejecutar las acciones de dibujo.

Partes más relevantes del código

Definiciones y directivas:

```
%{
    import java.lang.Math;
    import java.io.*;
    import java.util.StringTokenizer;
    import Configuracion.Configuracion;
%}

%token IF ELSE WHILE FOR COMP DIFERENTES MAY MEN MAYI MENI FNCT NUMBER VAR AND OR FUNC RETURN
PARAMETRO PROC
%right '='
%left '+' '-'
%left '*' '/'
%left ';'
%left COMP DIFERENTES MAY MEN MAYI MENI '!'
%left AND OR
%right RETURN
%%
```

- Se importan las clasesnecesarias.
- Se definen los tokens que el lexer debe reconocer (palabras clave, operadores, etc.).
- Se establecen las precedencias y asociatividades de los operadores. **Reglas de**

producción para list y linea:

```
list:
    | list '\n'
    | list linea '\n'
    ;

linea:
    exp ';' { $$ = $1; }
    | stmt { $$ = $1; }
    | linea exp ';' { $$ = $1; }
    | linea stmt { $$ = $1; }
    ;
```

Definen cómo se pueden combinar líneas de código y expresiones en listas de instrucciones.

Reglas de producción para exp:

```
exp:
    VAR {
        $$ = new ParserVal(maquina.agregarOperacion("varPush_Eval"));
        maquina.agregar($1.sval);
    }
    | '-' exp {
        $$ = new ParserVal(maquina.agregarOperacion("negativo"));
    }
    | NUMBER {
        $$ = new ParserVal(maquina.agregarOperacion("constPush"));
        maquina.agregar($1.dval);
    }
```

- Manejan variables, números y operaciones unarias.
- VAR: Empuja una variable en la pila.
- - exp: Realiza una operación de negación.
- NUMBER: Empuja un número constante en la pila.

Reglas de producciónparastmt:

```
stmt:
    if '(' exp stop ')' '{' linea stop '}' ELSE '{' linea stop '}' {
        $$ = $1;
        maquina.agregar($7.ival, $1.ival + 1);
        maquina.agregar($12.ival, $1.ival + 2);
        maquina.agregar(maquina.numeroDeElementos() - 1, $1.ival + 3);
    }
    | while '(' exp stop ')' '{' linea stop '}' stop {
        $$ = $1;
        maquina.agregar($7.ival, $1.ival + 1);
        maquina.agregar($10.ival, $1.ival + 2);
    }
    | for '(' instrucciones stop ';' exp stop ';' instrucciones stop ')' '{' linea stop '}'
stop {
        $$ = $1;
        maquina.agregar($6.ival, $1.ival + 1);
        maquina.agregar($9.ival, $1.ival + 2);
        maquina.agregar($13.ival, $1.ival + 3);
        maquina.agregar($16.ival, $1.ival + 4);
    }
```

- Manejan estructuras de control como if-else, while, y for.
- if-else: Agrega las posiciones de las instrucciones correspondientes a if, else, y la siguiente instrucción.
- while: Define el bucle while con su condición y cuerpo.
- for: Define elbucle for con sus inicializaciones, condición, ycuerpo.

Método insertarInstrucciones:

```
void insertarInstrucciones(){
    tablaDeSimbolos.insertar("TURN", new MaquinaDePila.Girar());
    tablaDeSimbolos.insertar("FORWARD", new MaquinaDePila.Avanzar());
    tablaDeSimbolos.insertar("COLOR", new MaquinaDePila.CambiarColor());
    tablaDeSimbolos.insertar("PenUP", new MaquinaDePila.SubirPincel());
    tablaDeSimbolos.insertar("PenDOWN", new MaquinaDePila.BajarPincel());
}
```

Inserta funciones específicas de dibujo en la tabla de símbolos.

Este archivo es esencial para definir cómo se interpreta y ejecuta el lenguaje Logos, especificando tanto la sintaxis como las acciones semánticas que deben tomarse al reconocer cada estructura del lenguaje.

Par.java

Explicación

El módulo Par.java define la clase Par, que es utilizada para representar un par de valores compuesto por un nombre (cadena de texto) y un objeto genérico. Esta clase es esencial para la gestión de la tabla de símbolos, ya que permite asociar nombres con objetos (que pueden ser variables, funciones, etc.) en el contexto del compilador.

Explicación de su unión con otros módulos

Par.java es utilizado principalmente en el módulo TablaDeSimbolos.java para almacenar y gestionar los pares de nombres y objetos. La tabla de símbolos usa instancias de Par para mapear nombres de variables y funciones a sus respectivos objetos y valores. Esto permite al compilador y a la máquina de pila acceder y manipular estos valores de manera eficiente durante la ejecución del programa.

Partes más relevantes del código

Clase Par:

Define dos atributos privados: nombre (de tipo String) y objeto (de tipo Object).

```
package Compilador;

public class Par {
    private String nombre;
    private Object objeto;

    public Par(String nombre, Object objeto) {
        this.nombre = nombre;
        this.objeto = objeto;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
```

```
    }

    public Object getObjeto() {
        return objeto;
    }

    public void setObjeto(Object objeto) {
        this.objeto = objeto;
    }
}
```

Constructor Par(String nombre, Object objeto):

Inicializa los atributos nombre y objeto con los valores proporcionados. **Métodos**

getNombreysetNombre:

- getNombre: Retorna el nombre del par.
- setNombre: Establece el nombre delpar.

MétodosgetObjetoysetObjeto:

- getObjeto: Retorna el objeto asociado con el nombre.
- setObjeto: Establece el objeto asociado con el nombre.

Este módulo es fundamental para la gestión de la tabla de símbolos, ya que permite almacenar y recuperar valores asociados a nombres, facilitando la manipulación de variables y funciones durante la ejecución del programa.

Parser.java

Explicación

El módulo Parser.java es la clase generada por BYACC (Berkeley YACC para Java) a partir del archivo P2.y. Esta clase define el parser que se utiliza para analizar y ejecutar el lenguaje Logos. Implementa las reglas de gramática y las acciones semánticas especificadas en P2.y, permitiendo la interpretación y ejecución de scripts escritos en Logos.

Explicación de su unión con otros módulos

MaquinaDePila.java: Utiliza la máquina de pila para ejecutar las instrucciones reconocidas por el parser.

TablaDeSimbolos.java: Gestiona variables y funciones definidas en los scripts.

Configuracion.java y Linea.java: Ejecuta acciones de dibujo basadas en las instrucciones del script.

Partes más relevantes del código

```
public class Parser {
    boolean  yydebug;
    int yynerrs;
    int yyerrflag;
    int yychar;

    final static int YYSTACKSIZE = 500;
    int statestk[] = new int[YYSTACKSIZE];
    int stateptr;
    int stateptrmax;
    int statemax;

    String yytext;
    ParserVal yyval;
    ParserVal yylval;
    ParserVal valstk[];
    int valptr;

    final static short IF = 257;
    final static short ELSE = 258;
    // ... other token definitions ...

    final static short yylhs[] = { ... };
    final static short yylen[] = { ... };
    final static short yydefred[] = { ... };
    final static short yysindex[] = { ... };
    final static short yyrindex[] = { ... };
    final static short yygindex[] = { ... };
    final static int YYTABLESIZE = 546;
    static short yytable[];
    static void yytable() { ... }
    static short yycheck[];
    static void yycheck() { ... }

    final static short YYFINAL = 1;
    final static short YYMAXTOKEN = 275;
    final static String yyname[] = { ... };
    final static String yyrule[] = { ... };

    // ... other parser code ...

    TablaDeSimbolos tablaDeSimbolos = new TablaDeSimbolos();
    MaquinaDePila maquina = new MaquinaDePila(tablaDeSimbolos);
    int i = 0;
    int j = 0;
    double[][] auxiliar;
    Funcion funcionAux;
    boolean huboError;

    String ins;
    StringTokenizer st;

    void yyerror(String s) {
        huboError = true;
        System.out.println("error:" + s);
        System.exit(0);
    }

    boolean newline;
    int yylex() {
        String s;
        int tok = 0;
        Double d;
```

```
if (!st.hasMoreTokens()) {
    if (!newline) {
        newline = true;
        return '\n';
    } else
        return 0;
}
s = st.nextToken();
try {
    d = Double.valueOf(s);
    yylval = new ParserVal(d.doubleValue());
    return NUMBER;
} catch (Exception e) { }
if (esVariable(s)) {
    if (s.equals("proc")) {
        return PROC;
    }
    if (s.charAt(0) == '$') {
        yylval = new ParserVal((int) Integer.parseInt(s.substring(1)));
        return PARAMETRO;
    }
    if (s.equals("return")) {
        return RETURN;
    }
    if (s.equals("func")) {
        return FUNC;
    }
    if (s.equals("if")) {
        return IF;
    }
    if (s.equals("else")) {
        return ELSE;
    }
    if (s.equals("while")) {
        return WHILE;
    }
    if (s.equals("for")) {
        return FOR;
    }
    boolean esFuncion = false;
    Object objeto = tablaDeSimbolos.encontrar(s);
    if (objeto instanceof Funcion) {
        funcionAux = (Funcion) objeto;
        yylval = new ParserVal(objeto);
        esFuncion = true;
        return FNCT;
    }
    if (!esFuncion) {
        yylval = new ParserVal(s);
        return VAR;
    }
} else {
    if (s.equals("==")) {
        return COMP;
    }
    if (s.equals("!=")) {
        return DIFERENTES;
    }
    if (s.equals("<")) {
        return MEN;
    }
    if (s.equals("<=")) {
        return MENI;
    }
    if (s.equals(">")) {
```

```
                return MAY;
            }
            if (s.equals(">=")) {
                return MAYI;
            }
            if (s.equals("&&")) {
                return AND;
            }
            if (s.equals("||")) {
                return OR;
            }
            tok = s.charAt(0);
        }
        return tok;
    }

    String reservados[] = { ">=", "&&", "||", "<=", "==", "!=", "=", "{", "}", ",", "*", "+",
"-", "(", ")", "|", "[", "]", ";", "!", "<", ">", "/" };

    public String ajustarCadena(String cadena) {
        String nueva = "";
        boolean encontrado = false;
        for (int i = 0; i < cadena.length() - 1; i++) {
            encontrado = false;
            for (int j = 0; j < reservados.length; j++) {
                if (cadena.substring(i, i + reservados[j].length()).equals(reservados[j])) {
                    i += reservados[j].length() - 1;
                    nueva += " " + reservados[j] + " ";
                    encontrado = true;
                    break;
                }
            }
            if (!encontrado)
                nueva += cadena.charAt(i);
        }
        nueva += cadena.charAt(cadena.length() - 1);
        return nueva;
    }

    boolean esVariable(String s) {
        boolean cumple = true;
        for (int i = 0; i < reservados.length; i++)
            if (s.equals(reservados[i]))
                cumple = false;
        return cumple;
    }

    public void insertarInstrucciones() {
        tablaDeSimbolos.insertar("TURN", new MaquinaDePila.Girar());
        tablaDeSimbolos.insertar("FORWARD", new MaquinaDePila.Avanzar());
        tablaDeSimbolos.insertar("COLOR", new MaquinaDePila.CambiarColor());
        tablaDeSimbolos.insertar("PenUP", new MaquinaDePila.SubirPincel());
        tablaDeSimbolos.insertar("PenDOWN", new MaquinaDePila.BajarPincel());
    }

    public boolean compilar(String codigo) {
        st = new StringTokenizer(ajustarCadena(codigo));
        newline = false;
        yyparse();
        return !huboError;
    }

    public boolean ejecutarSiguiente() {
        return maquina.ejecutarSiguiente();
    }
```

```
    public Configuracion getConfiguracion() {
        return maquina.getConfiguracion();
    }

    public void limpiar() {
        tablaDeSimbolos = new TablaDeSimbolos();
        insertarInstrucciones();
        maquina = new MaquinaDePila(tablaDeSimbolos);
    }

    public Configuracion ejecutar() {
        maquina.ejecutar();
        return maquina.getConfiguracion();
    }

    void dotest() {
        insertarInstrucciones();
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            huboError = false;
            try {
                ins = ajustarCadena(in.readLine());
            } catch (Exception e) { }
            st = new StringTokenizer(ins);
            newline = false;

            yyparse();
            if (!huboError)
                maquina.ejecutar();
        }
    }

    public static void main(String args[]) {
        Parser par = new Parser(false);
        par.dotest();
    }

    int yyparse() {
        // Parsing logic generated by BYACC
        // ...
    }

    void yylexdebug(int state, int ch) {
        String s = null;
        if (ch < 0) ch = 0;
        if (ch <= YYMAXTOKEN)
            s = yyname[ch];
        if (s == null)
            s = "illegal-symbol";
        debug("state " + state + ", reading " + ch + " (" + s + ")");
    }

    // Other generated methods for parsing
    // ...
}
```

Atributos principales:

- yydebug, yynerrs, yyerrflag, yychar: Variables de control del parser.
- statestk, stateptr, stateptrmax, statemax: Pila de estados y punteros para el control del parser.

- yytext, yyval, yylval, valstk, valptr: Variables para gestionar los valores semánticos durante el análisis.
- TablaDeSimbolos tablaDeSimbolos: Instancia de la tabla de símbolos.
- MaquinaDePila maquina: Instancia de la máquina de pila. **Método**

yylex():

- Realiza el análisis léxico, obteniendo tokens del código fuente.
- Convierte cadenas de texto en tokens reconocidos (como números, variables, operadores, etc.).

Método insertarInstrucciones():

Inserta las funciones específicas de dibujo (TURN, FORWARD, COLOR, PenUP, PenDOWN) en la tabla de símbolos.

Método compilar(String codigo):

- Compila el código fuente dado, utilizando el tokenizer yel parser generado.
- Retorna true si la compilación fue exitosa, false en caso contrario. **Método**

ejecutar():

- Ejecuta el código compilado utilizando la máquina de pila.
- Retorna la configuración resultante después de la ejecución. **Método**

dotest():

Método de prueba que permite la ejecución interactiva de código Logos. **Uniónconotros**

módulos:

- TablaDeSimbolos: Maneja la inserción y búsqueda de variablesy funciones.
- MaquinaDePila: Ejecuta las instrucciones compiladas.
- Configuracion: Configura el estado de dibujo según las instrucciones ejecutadas.

Este módulo es crucial para la interpretación y ejecución de scripts escritos en Logos, permitiendo la traducción de código fuente en acciones concretas mediante el uso de la máquina de pila y la tabla de símbolos.

ParserVal.java

Explicación

El módulo ParserVal.java define la clase ParserVal, que es utilizada como una estructura de datos para almacenar los valores semánticos de los tokens durante el proceso de análisis sintáctico. Esta clase es el equivalente a la unión (union) en YACC para C, proporcionando diferentes tipos de datos que pueden ser utilizados por el parser.

Explicación de su unión con otros módulos

ParserVal.java es utilizado directamente por el módulo Parser.java, que es el parser generado por BYACC. Cada token y regla de producción en el archivo P2.y puede devolver un valor semántico que se almacena en una instancia de ParserVal. Estos valores se utilizan en las acciones semánticas para realizar cálculos, almacenar resultados intermedios, y manejar la ejecución del lenguaje Logos.

Partes más relevantes del código

```
package Compilador;
public class ParserVal {
    public int ival; // Tipo de datos entero
    public double dval; // Tipo de dato double
    public String sval; // Tipo de dato string
    public Object obj; // Tipo de dato objeto
    public ParserVal(int val) {
        ival = val;
    }
    public ParserVal(double val) {
        dval = val;
    }
    public ParserVal(String val) {
        sval = val;
    }
    public ParserVal(Object val) {
        obj = val;
    }
}
```

Atributos de la clase ParserVal:

- int ival: Almacena valores enteros.
- doubledval: Almacena valores de tipo double.
- String sval: Almacena valores de tipo String.
- Object obj: Almacena valores de tipo Object.

Constructores:

- ParserVal(): Constructor por defecto que inicializa la instancia sin valores.
- ParserVal(int val): Inicializa la instancia con un valor entero.
- ParserVal(double val): Inicializa la instancia con un valor double.
- ParserVal(String val): Inicializa la instancia con un valor String.
- ParserVal(Object val): Inicializa la instancia con un valor Object.

Este módulo es crucial para la gestión de los valores semánticos durante el análisis sintáctico, permitiendo al parser almacenar y manipular diferentes tipos de datos de manera flexible.

TablaDeSimbolos.java

Explicación

El módulo TablaDeSimbolos.java define la clase TablaDeSimbolos, que se encarga de gestionar la tabla de símbolos del compilador. Una tabla de símbolos es una estructura de datos utilizada por los compiladores para almacenar información sobre las variables, funciones y otros identificadores encontrados en el código fuente. Esta clase permite insertar, buscar y modificar los símbolos y sus valores asociados.

Explicación de su unión con otros módulos

TablaDeSimbolos.java es utilizada principalmente por el módulo Parser.java y MaquinaDePila.java para gestionar los símbolos (variables, funciones) durante el análisis sintáctico y la ejecución. Cada símbolo se almacena como un par de nombre y objeto (Par), lo que facilita la asociación de identificadores con sus respectivos valores o funciones.

Partes más relevantes del código

```
package Compilador;
import java.util.ArrayList;

public class TablaDeSimbolos {
    ArrayList<Par> simbolos;

    public TablaDeSimbolos() {
        simbolos = new ArrayList<Par>();
    }

    public Object encontrar(String nombre) {
        for (int i = 0; i < simbolos.size(); i++)
            if (nombre.equals(simbolos.get(i).getNombre()))
                return simbolos.get(i).getObjeto();
        return null;
    }

    public boolean insertar(String nombre, Object objeto) {
        Par par = new Par(nombre, objeto);
        for (int i = 0; i < simbolos.size(); i++)
            if (nombre.equals(simbolos.get(i).getNombre())) {
                simbolos.get(i).setObjeto(objeto);
                return true;
            }
        simbolos.add(par);
        return false;
    }

    public void imprimir() {
        for (int i = 0; i < simbolos.size(); i++) {
            System.out.println(simbolos.get(i).getNombre()          +
simbolos.get(i).getObjeto().toString());
        }
    }
}
```

Atributos de la clase TablaDeSimbolos:

ArrayList<Par> simbolos: Lista que almacena los pares de nombres y objetos (símbolos) gestionados por la tabla de símbolos.

Constructor TablaDeSimbolos():

Inicializa la lista de símbolos. **Método**

encontrar(String nombre):

- Busca un símbolo por su nombre en la lista de símbolos.
- Retorna el objeto asociado con el nombre si se encuentra, de lo contrario retorna null.

Método insertar(String nombre, Object objeto):

- Inserta un nuevo símbolo en la tabla de símbolos.
- Si el símbolo ya existe, actualiza su objeto asociado.
- Retorna true si el símbolo fue actualizado, false si fue insertado. **Método**

imprimir():

Imprime todos los símbolos y sus valores enla consola.

Este módulo es fundamental para la gestión de los identificadores y sus valores asociados durante el análisis y la ejecución del código fuente. Permite la inserción, búsqueda y modificación eficiente de símbolos, facilitando la correcta interpretación y ejecución del lenguaje Logos.

Módulos de configuración.

Configuracion.java

Explicación

El módulo Configuracion.java define la clase Configuracion, que gestiona el estado actual de la configuración del dibujo en el lenguaje Logos. Esta clase maneja la posición, ángulo y color del trazo, así como las líneas dibujadas. Es esencial para mantener el estado de la "tortuga" en términos de su posición y dirección, así como el registro de las líneas dibujadas.

Explicación de su unión con otros módulos

Configuracion.java es utilizada por:

- MaquinaDePila.java: Modifica la configuración durante la ejecución de comandos de dibujo.
- PanelDeDibujo.java: Utiliza la configuración para renderizar el estado actual del dibujo en el panel.

Partes más relevantes del código

```
package Configuracion;
import java.awt.Color;
import java.util.ArrayList;

public class Configuracion {
    private final ArrayList<Linea> lineas;
    private double x;
    private double y;
    private int angulo;
    private Color color;

    public Configuracion() {
        x = 0.0;
        y = 0.0;
        lineas = new ArrayList<>();
    }

    public ArrayList<Linea> getLineas() { return lineas; }
    public double getX() { return x; }
    public double getY() { return y; }
    public int getAngulo() { return angulo; }
    public Color getColor() { return color; }

    public void setAngulo(int angulo) { this.angulo = angulo; }
    public void setColor(Color color) { this.color = color; }

    public void setPosicion(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public void agregarLinea(Linea linea) {
        lineas.add(linea);
    }

    @Override
    public String toString() {
        StringBuilder valor = new StringBuilder();

        for (Linea linea : lineas) valor.append(linea).append(", ");

        valor.append("x:").append(x).append(" y:").append(y).append(" angulo:
").append(angulo);
        return valor.toString();
    }
}
```

Atributos de la clase Configuracion:

- ArrayList<Linea> lineas: Lista que almacenalas líneas dibujadas.
- double x, y: Coordenadas de la posición actual.
- int angulo: Ángulo de dirección actual.
- Color color: Color actual del trazo.

Constructor Configuracion():

Inicializa la posición en (0.0, 0.0) ycrea una lista vacía de líneas. **Métodos get y**

set:

Proveen acceso y modificación a las propiedades de la configuración (posición, ángulo, color).

Método agregarLinea(Linea linea):

Añade una línea a la lista de líneas. **Método**

toString():

Retorna una representación en cadena del estado actual de la configuración, incluyendo todas las líneas y la posición y ángulo actuales.

Este módulo es esencial para mantener y gestionar el estado del dibujo en el lenguaje Logos, permitiendo modificar la posición, ángulo y color de la "tortuga", así como registrar las líneas dibujadas.

Linea.java

Explicación

El módulo Linea.java define la clase Linea, que representa una línea en el panel de dibujo del lenguaje Logos. Cada instancia de Linea contiene las coordenadas de los puntos inicial y final de la línea, así como su color. Esta clase se utiliza para almacenar y gestionar las líneas dibujadas en el panel de dibujo.

Explicación de su unión con otros módulos

Linea.java es utilizada por:

- Configuracion.java: Almacena las líneas dibujadas en una lista y proporciona métodos para añadir nuevas líneas.
- PanelDeDibujo.java: Utiliza las instancias de Linea para renderizar las líneas en el panel de dibujo.

Partes más relevantes del código

```
package Configuracion;
import java.awt.Color;

public class Linea {
    int x0;
    int y0;
    int x1;
    int y1;
    Color color;

    public Linea(int x0, int y0, int x1, int y1, Color color) {
        this.x0 = x0;
        this.y0 = y0;
        this.x1 = x1;
        this.y1 = y1;
        this.color = color;
    }

    public int getX0() { return x0; }
    public int getY0() { return y0; }
    public int getX1() { return x1; }
    public int getY1() { return y1; }
    public Color getColor() {
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
    }

    @Override
    public String toString() {
        return "(" + x0 + "," + y0 + "," + x1 + "," + y1 + ")";
    }
}
```

Atributos de la clase Linea:

- int x0, y0: Coordenadas del punto inicial de la línea.
- int x1, y1: Coordenadas del punto final de la línea.
- Colorcolor: Color de la línea.

Constructor Linea(int x0, int y0, int x1, int y1, Color color): Inicializa los

atributos de la línea con los valores proporcionados. **Métodos get y set:**

Proveenaccesoymodificación alas propiedades delalínea (coordenadas ycolor). **Método**

toString():

Retorna una representación en cadena de la línea, mostrando las coordenadas de los puntos inicial y final.

Este módulo es fundamental para representar y gestionar las líneas dibujadas en el panel de dibujo, proporcionando una estructura clara y accesible para almacenar las propiedades de cada línea.

Módulo logos.

Logos.java

Explicación

El módulo Logos.java define la clase Logos, que contiene el método principal (main) para iniciar la aplicación de dibujo basada en el lenguaje Logos. Este módulo sirve como punto de entrada para la ejecución del programa y se encarga de crear la ventana principal de la aplicación.

Explicación de su unión con otros módulos

Logos.java se relaciona con el módulo VentanaPrincipal.java del paquete Vista. Al iniciar la aplicación, crea una instancia de VentanaPrincipal, que configura la interfaz gráfica del usuario (GUI) para permitir la escritura y ejecución de scripts en Logos.

Partes más relevantes del código

```
package Logos;
import Vista.VentanaPrincipal;

public class Logos {
    public static void main(String args[]) {
        new VentanaPrincipal();
    }
}
```

Método main(String args[]):

- Este método es el punto de entrada de la aplicación Java.
- Crea una nueva instancia de VentanaPrincipal, lo que inicia la interfaz gráfica de la aplicación.

Este módulo es esencialmente el iniciador de la aplicación, lanzando la GUI y permitiendo al usuario interactuar con el entorno de programación Logos.

Módulos de vista.

PanelDeDibujo.java

Explicación

El módulo PanelDeDibujo.java define la clase PanelDeDibujo, que es un componente gráfico utilizado para renderizar las figuras y líneas dibujadas mediante el lenguaje Logos. Este panel se encarga de visualizar el estado actual de la "tortuga" y las líneas dibujadas en el panel de dibujo.

Explicación de su unión con otros módulos

PanelDeDibujo.java se relaciona con los siguientes módulos:

Configuracion.java: Proporciona la configuración actual (posición, ángulo, color) de la "tortuga" y las líneas a dibujar.

VentanaPrincipal.java: Contiene una instancia de PanelDeDibujo para mostrar el área de dibujo en la interfaz gráfica.

Partes más relevantes del código

```
public class PanelDeDibujo extends JPanel {

    Configuracion configuracion;

    public PanelDeDibujo() {
        configuracion = new Configuracion();
    }

    public void setConfiguracion(Configuracion configuracion) {
        this.configuracion = configuracion;
    }

    @Override
    public void paint(Graphics g) {
        super.paint(g);
        ArrayList<Linea> lineas = configuracion.getLineas();

        for (int i = 0; i < lineas.size(); i++) {
            int x0 = (Propiedades.PANEL_DE_DIBUJO_ANCHO/2) + lineas.get(i).getX0();
            int y0 = (Propiedades.PANEL_DE_DIBUJO_LARGO/2) - lineas.get(i).getY0();
            int x1 = (Propiedades.PANEL_DE_DIBUJO_ANCHO/2) + lineas.get(i).getX1();
            int y1 = (Propiedades.PANEL_DE_DIBUJO_LARGO/2) - lineas.get(i).getY1();
            g.setColor(lineas.get(i).getColor());
            g.drawLine(x0, y0, x1, y1);
        }

        g.setColor(Color.BLACK);
        Polygon triangulo = triangulo(configuracion.getX(), configuracion.getY(),
configuracion.getAngulo());
        g.drawPolygon(triangulo);
        g.fillPolygon(triangulo);
    }

    public Polygon triangulo(double x, double y, int angulo) {
```

```
        int[] xs = new int[3];
        int[] ys = new int[3];

        xs[0] = (Propiedades.PANEL_DE_DIBUJO_ANCHO/2) + (int) x;
        ys[0] = (Propiedades.PANEL_DE_DIBUJO_LARGO/2) - (int) y;
         xs[1]   =   (Propiedades.PANEL_DE_DIBUJO_ANCHO/2)   +   (int)   (x   -   10   *
Math.cos(Math.toRadians(angulo + 20)));
        ys[1]   =   (Propiedades.PANEL_DE_DIBUJO_LARGO/2)     -   (int)   (y   -   10   *
Math.sin(Math.toRadians(angulo + 20)));
         xs[2]   =   (Propiedades.PANEL_DE_DIBUJO_ANCHO/2)   +   (int)   (x   -   10   *
Math.cos(Math.toRadians(angulo - 20)));
        ys[2]   =   (Propiedades.PANEL_DE_DIBUJO_LARGO/2)     -   (int)   (y   -   10   *
Math.sin(Math.toRadians(angulo - 20)));

        return new Polygon(xs, ys, 3);
    }
}
```

Atributos de la clase PanelDeDibujo:

Configuracion configuracion: La configuración actual de la "tortuga" y las líneas a dibujar.

Constructor PanelDeDibujo():

Inicializa la configuración con una nueva instancia de Configuracion. **Método**

setConfiguracion(Configuracion configuracion):

Establece la configuración actual del panel de dibujo. **Método**

paint(Graphics g):

- Sobrescribe el método paint de JPanel para dibujar las líneas y la "tortuga" en el panel.
- Recorre la lista de líneas de la configuración y las dibuja en el panel.
- Dibuja la "tortuga" como un triángulo en la posición y ángulo actuales. **Método**

triangulo(double x, double y, int angulo):

- Calcula las coordenadas del triángulo que representa la "tortuga" en función de su posición y ángulo.
- Retorna un objeto Polygon con las coordenadas del triángulo.

Este módulo es crucial para la visualización gráfica del estado del dibujo en el lenguaje Logos, proporcionando una representación visual clara de las líneas dibujadas y la posición de la "tortuga".

Propiedades.java

Explicación

El módulo Propiedades.java define una clase con constantes estáticas que se utilizan para configurar las propiedades del panel de dibujo. Este módulo

proporciona valores constantes que se utilizan en el diseño de la interfaz gráfica para establecer el tamaño del área de dibujo.

Explicación de su unión con otros módulos

Propiedades.java se utiliza en:

- PanelDeDibujo.java: Utiliza las constantes para establecer el tamaño y las coordenadas de las líneas y la tortuga en el panel de dibujo.
- VentanaPrincipal.java: Utiliza las constantes para establecer el tamaño del panel de dibujo dentro de la ventana principal.

Partes más relevantes del código

```
package Vista;

public class Propiedades {
    public static final int PANEL_DE_DIBUJO_ANCHO = 710;
    public static final int PANEL_DE_DIBUJO_LARGO = 650;
}
```

Constantes estáticas:

- PANEL_DE_DIBUJO_ANCHO: Define el ancho del panel de dibujo (710 píxeles).
- PANEL_DE_DIBUJO_LARGO: Defineellargodelpaneldedibujo(650píxeles).

Estas constantes se utilizan para asegurar que el panel de dibujo tenga dimensiones consistentes en toda la aplicación. Al definirlas como constantes, se facilita la modificación de las dimensiones del panel de dibujo en un solo lugar si es necesario realizar cambios en el futuro.

Este módulo es esencial para mantener la consistencia del tamaño del panel de dibujo y garantizar que las dimensiones del área de dibujo sean adecuadas para la representación gráfica.

VentanaPrincipal.java

Explicación

El módulo VentanaPrincipal.java define la clase VentanaPrincipal, que es la ventana principal de la aplicación de dibujo basada en el lenguaje Logos. Esta clase configura la interfaz gráfica del usuario (GUI), permitiendo la escritura, carga, y ejecución de scripts en Logos, así como la visualización del dibujo generado.

Explicación de su unión con otrosmódulos

VentanaPrincipal.java se relaciona con:

- PanelDeDibujo.java: UtilizaunainstanciadePanelDeDibujoparamostrarel área de dibujo.
- Parser.java: Utiliza una instancia de Parser para compilar y ejecutar el código ingresado por el usuario.
- Propiedades.java: Utiliza las constantes definidas para establecer el tamaño del panel de dibujo.

Partesmásrelevantes del código

```
import Compilador.Parser;

public class VentanaPrincipal extends JFrame {
    private final JTextArea areaDeCodigo;
    private final PanelDeDibujo panelDeDibujo;
    private final JButton ejecutar;
    Parser parser;

    // Constructor de la ventana principal
    public VentanaPrincipal() {
        super("Logos");

        // Configurar el fondo del JFrame
        getContentPane().setBackground(new Color(0, 128, 0)); // Verde fuerte

        parser = new Parser();
        parser.insertarInstrucciones();

        // Área de texto para el código con estilo personalizado
        areaDeCodigo = new JTextArea(20, 20);
        areaDeCodigo.setBackground(new Color(204, 204, 204)); // Color de fondo gris claro
        areaDeCodigo.setForeground(new Color(102, 0, 102)); // Color de texto púrpura oscuro
        areaDeCodigo.setFont(new Font("Monospaced", Font.BOLD, 18)); // Fuente monospaced,
negrita, tamaño 18
        JScrollPane scrollCodigo = new JScrollPane(areaDeCodigo);
        scrollCodigo.setBounds(10, 10, 330, 535);
        add(scrollCodigo);

        // Botón para cargar un archivo de texto con estilo personalizado
        JButton Archivo_sel = new JButton("Carga un archivo de texto");
        Archivo_sel.setBounds(10, 560, 330, 40);
        Archivo_sel.setVisible(true);
        Archivo_sel.setBackground(new Color(204, 255, 204)); // Color de fondo verde claro
        Archivo_sel.setForeground(new Color(102, 0, 102)); // Color de texto púrpura oscuro
        Archivo_sel.setFont(new Font("Monospaced", Font.BOLD, 18)); // Fuente monospaced,
negrita, tamaño 18
        add(Archivo_sel);

        // Configuración del ícono de la ventana
        this.setIconImage(new ImageIcon(getClass().getResource("/Imgs/2.jpeg")).getImage());

        // Acción para el botón de carga de archivo
        Archivo_sel.addActionListener(event->{
            JFileChooser chooser = new
JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());
            chooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
            FileNameExtensionFilter filter = new FileNameExtensionFilter("TEXT FILES", "txt",
"text");
```

```
            chooser.setFileFilter(filter);
            int returnVal = chooser.showOpenDialog(this);

            if(returnVal == JFileChooser.APPROVE_OPTION){
                File file = chooser.getSelectedFile();
                try{
                    BufferedReader in;
                    in = new BufferedReader(new FileReader(file));
                    String linea = in.readLine();
                    if(linea != null){
                        areaDeCodigo.setText(linea);
                        while ((linea = in.readLine()) != null){
                            areaDeCodigo.setText(areaDeCodigo.getText() + "\n" + linea);
                        }
                    }

                }catch(Exception e){
                    e.printStackTrace();
                }
            }
        });
        add(Archivo_sel);

        // Panel de dibujo para mostrar los gráficos
        panelDeDibujo = new PanelDeDibujo();
        panelDeDibujo.setBounds(350,              10,              Propiedades.PANEL_DE_DIBUJO_ANCHO,
Propiedades.PANEL_DE_DIBUJO_LARGO);
        panelDeDibujo.setBackground(Color.WHITE);
        add(panelDeDibujo);

        // Agregar la imagen debajo del panel de dibujo
        ImageIcon logoIcon = new ImageIcon("/Imgs/logo_writer.png");
        JLabel logoLabel = new JLabel(logoIcon);
        logoLabel.setBounds(350,          Propiedades.PANEL_DE_DIBUJO_LARGO          +          20,
logoIcon.getIconWidth(), logoIcon.getIconHeight());
        add(logoLabel);

        // Botón para ejecutar el código con estilo personalizado
        ImageIcon icon1 = new ImageIcon(getClass().getResource("/Imgs/1.jpeg"));
        Image img1 = icon1.getImage();
        Image imgaux = img1.getScaledInstance(30, 30, java.awt.Image.SCALE_SMOOTH);
        ejecutar = new JButton(new ImageIcon(imgaux));
        ejecutar.setBounds(10, 620, 50, 40);
        ejecutar.setBackground(new Color(204, 255, 204)); // Color de fondo verde claro
        ejecutar.setForeground(new Color(102, 0, 102));  // Color de texto púrpura oscuro
        ejecutar.setFont(new Font("Monospaced", Font.BOLD, 18)); // Fuente monospaced, negrita,
tamaño 18
        ejecutar.addActionListener(event -> {
            parser.limpiar();
            if (parser.compilar(areaDeCodigo.getText()))
                panelDeDibujo.setConfiguracion(parser.ejecutar());
            else {
                parser = new Parser();
                parser.insertarInstrucciones();
                panelDeDibujo.setConfiguracion(parser.getConfiguracion());
            }
            panelDeDibujo.repaint();
        });
        add(ejecutar);

        // Botón para limpiar el área de código con estilo personalizado
        ImageIcon iconClr = new ImageIcon(getClass().getResource("/Imgs/clear.png"));
        Image imgClr = iconClr.getImage();
        Image imgauxClr = imgClr.getScaledInstance(30, 30, java.awt.Image.SCALE_SMOOTH);
        JButton clr = new JButton(new ImageIcon(imgauxClr));
```

```
        clr.setBounds(65, 620, 50, 40);
        clr.setBackground(new Color(204, 255, 204)); // Color de fondo verde claro
        clr.setForeground(new Color(102, 0, 102));  // Color de texto púrpura oscuro
        clr.setFont(new Font("Monospaced", Font.BOLD, 18)); // Fuente monospaced, negrita, tamaño
18
        clr.addActionListener(event -> {
            parser.limpiar();
            areaDeCodigo.setText("");
        });
        add(clr);

        // Configuración inicial de la ventana
        setLayout(null);
        setBounds(50, 50, 1100, 730); // Ajustar el tamaño de la ventana
        setVisible(true);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Atributos de la clase VentanaPrincipal:

- JTextArea areaDeCodigo: Área de texto donde el usuario ingresa el código.
- PanelDeDibujopanelDeDibujo: Panelparadibujarlasfigurasgeneradaspor el código.
- JButton ejecutar: Botón para ejecutar el código.
- Parser parser: Instancia del parser paracompilar y ejecutar elcódigo.

Constructor VentanaPrincipal():

- Configura la ventana principal, establece el fondo, y crea y configura los componentes GUI.
- Inicializa el parser y el área de código.
- Añade botones para cargar archivos, ejecutar código, y limpiar el área de código.
- Configura el panel dedibujo y añade una imagen debajo de él.

Configuración del botón de carga de archivo:

Permite al usuario seleccionar un archivo de texto para cargar el código en el área de código.

Configuración del botón de ejecución:

- Compila y ejecuta el código ingresado en el área de código.
- Actualiza el panel de dibujo con la nueva configuración generada por el código.

Configuracióndel botón de limpieza: Limpia el área

de código y resetea el parser.

Este módulo es crucial para la interacción del usuario con la aplicación, proporcionando una interfaz gráfica para escribir, cargar, ejecutar, y visualizar código en el lenguaje Logos.

Funcionamiento del código.

El código proporcionado es parte de una aplicación de dibujo basada en el lenguaje Logos. Logos es un lenguaje gráfico inspirado en el lenguaje Logo, utilizado para enseñar conceptos de programación a través de la manipulación gráfica de una "tortuga" que dibuja en una pantalla. La aplicación está compuesta por varios módulos interconectados que juntos forman un compilador e intérprete para Logos.

Conceptos Clave de Compiladores en el Código

Análisis Léxico y Sintáctico

Análisis Léxico: La clase P2.y y el método yylex en el parser se encargan de dividir el código fuente en tokens, que son las unidades léxicas del lenguaje.

Análisis Sintáctico: La clase Parser.java, generada por BYACC, analiza la estructura gramatical de los tokens para asegurarse de que cumplen con las reglas del lenguaje Logos. Define cómo se deben interpretar y organizar las expresiones, comandos y funciones.

Tabla de Símbolos

La TablaDeSimbolos es una estructura de datos crucial que mantiene un registro de todas las variables y funciones definidas en el programa Logos. Esto permite la búsqueda eficiente y la asignación de valores durante la ejecución del programa.

Máquina de Pila

La MaquinaDePila simula una máquina virtual que ejecuta instrucciones almacenadas en una pila. Esta máquina maneja operaciones aritméticas, control de flujo (como bucles y condicionales), y la gestión de funciones.

Interfaz Gráfica del Usuario (GUI)

La VentanaPrincipal y el PanelDeDibujo forman la interfaz gráfica que permite a los usuarios escribir, cargar, y ejecutar código Logos. El PanelDeDibujo es responsable de renderizar las figuras dibujadas según las instrucciones del código.

Resumen del Funcionamiento del Código

Interfaz de Usuario

El usuario ingresa código Logos en un área de textodentro dela VentanaPrincipal.

El usuario puede cargar archivos de texto que contienen código Logos y ejecutar el código directamente desde la interfaz gráfica.

Compilación y Ejecución

Al compilar el código, el Parser analiza el código fuente y lo traduce en instrucciones ejecutables por la MaquinaDePila.

Las instrucciones se almacenan en una pila, y la MaquinaDePila ejecuta estas instrucciones, manipulando las variables y llamando a funciones según sea necesario.

Visualización

Las instrucciones que afectan la posición, ángulo, y color de la "tortuga" generan líneas que se dibujan en el PanelDeDibujo.

La Configuracion mantiene el estado actual del dibujo, incluyendo todas las líneas trazadas y laposición de la "tortuga".

Resumen de losmódulos.

1. Funcion.java: Interfaz para definir el método ejecutar, utilizado por las funciones en la máquina de pila.
2. MaquinaDePila.java: Implementa la máquina de pila para ejecutar instrucciones y operaciones, y contiene la lógica para manejar variables, operaciones aritméticas y de control de flujo.
3. Marco.java: Define un marco de función para la máquina de pila, almacenando parámetros y puntos de retorno.
4. P2.y: Archivo Yacc que define la gramática y las acciones semánticas para el lenguaje Logos, utilizado por el parser.
5. Par.java: Clase que asocia un nombre con un objeto, utilizada en la tabla de símbolos.
6. Parser.java: Clase generada por BYACC para el análisis sintáctico y la ejecución de instrucciones del lenguaje Logos.
7. ParserVal.java: Clase que actúa como una unión en YACC para almacenar diferentes tipos de datos en el análisis sintáctico.
8. TablaDeSimbolos.java: Implementa una tabla de símbolos para almacenar y buscar variables y funciones por nombre.

9. Configuracion.java: Clase que almacena la configuración actual del dibujo, incluyendo posición, ángulo, color y las líneas dibujadas.
10. Linea.java: Clase que representa una línea dibujada en el panel de dibujo, con coordenadas y color.
11. Logos.java: Clase principal que inicia la aplicación creando la ventana principal.
12. PanelDeDibujo.java: Clase que implementa el panel de dibujo para renderizar las líneas y la "tortuga" en la interfaz gráfica.
13. Propiedades.java: Clase que define constantes estáticas para el tamaño del panel de dibujo.
14. VentanaPrincipal.java: Clase que crea y configura la ventana principal de la aplicación, incluyendo el área de código, el panel de dibujo, y los botones de control.

Conclusión.

El desarrollo de este compilador e intérprete para el lenguaje Logos representa una valiosa aplicación práctica de los principios esenciales en la construcción de compiladores. A lo largo del proyecto, se abordan etapas clave como el análisis léxico, sintáctico y la ejecución mediante una máquina de pila, demostrando cómo diversas estructuras de datos y técnicas se integran para dar vida a una herramienta funcional.

Además, la implementación de una interfaz gráfica mejora significativamente la experiencia del usuario, haciendo que el proceso de programación sea más comprensible y atractivo, especialmente para quienes están dando sus primeros pasos en este campo. Este enfoque visual no solo facilita la enseñanza de la lógica de programación, sino que también introduce de manera accesible los fundamentos del diseño e implementación de lenguajes.