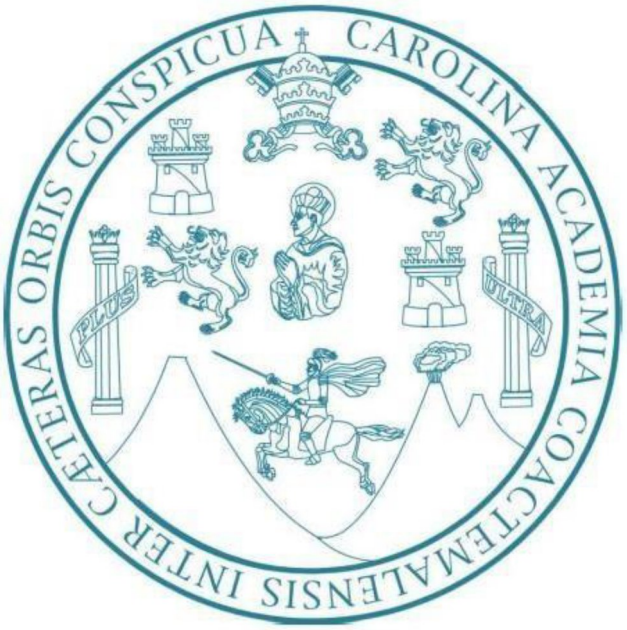
Universidad de San Carlos de Guatemala
División de Ciencias de la Ingeniería
Organización de Lenguajes y Compiladores 1



MANUAL TÉCNICO
GESTOR DE GRAFICADOR DE FIGURAS GEOMTRICAS

Estudiante:
Carné: 202132299
Nombre: Gérman Alex Ramirez Límatuj

GRAFICADOR DE FIGURAS GEOMTRICAS

La siguiente documentación muestra la estructura de código en la cual se construyo la aplicación, se conforma de la siguiente manera:

Paquete src/main/cup

El archivo CUP define un parser (analizador sintáctico) que interpreta un lenguaje específico diseñado para graficar figuras geométricas y aplicar animaciones a estas figuras. El propósito principal de este archivo es traducir una secuencia de tokens (generada por un lexer o analizador léxico) en una estructura lógica que represente un programa o conjunto de instrucciones. Este proceso es clave para convertir las entradas textuales en acciones programáticas que el sistema puede ejecutar.

Componentes Principales de la Gramática

1. **Terminales**:
   - Los terminales son los tokens que el parser recibe del lexer. Estos incluyen:
   - Palabras clave: `GRAFICAR`, `CIRCULO`, `CUADRADO`, `RECTANGULO`, `LINEA_OBJ`, `POLIGONO`, `ANIMAR`, `OBJETO`, `ANTERIOR`, `CURVA`, `LINEA`.
   - Operadores: `SUMA`, `RESTA`, `MULTIPLICACION`, `DIVISION`.
   - Paréntesis y comas: `PAR_IZQ`, `PAR_DER`, `COMA`.
   - Identificadores (`ID`), números (`NUMERO`), y colores (`AZUL`, `ROJO`, etc.).
1. **No Terminales**:
   - Los no terminales son combinaciones de terminales que forman estructuras más complejas en la gramática:
   - `programa`: Representa un conjunto de instrucciones que conforman un programa completo.
   - `instruccion`: Representa una acción específica que el programa puede realizar, como graficar una forma o animar un objeto.
   - `forma`: Representa las diferentes formas geométricas que pueden ser graficadas.
   - `animar`: Representa la instrucción para animar una figura.
   - `color`: Define los colores disponibles para las figuras.
   - `expresion`: Maneja las operaciones aritméticas dentro de las instrucciones.
1. **Precedencia de Operadores**:
   - CUP permite definir la precedencia de los operadores para resolver ambigüedades en las expresiones aritméticas. Aquí se define que `SUMA` y `RESTA` tienen menor precedencia que `MULTIPLICACION` y `DIVISION`.

Reglas de Producción y Análisis

1. **`inicial ::= programa;`**:
   - Esta regla indica que el análisis sintáctico comenzará con un `programa`, que es la estructura raíz del lenguaje. Una vez que se haya procesado todo el programa, el parser se detiene.
1. **`programa ::= instruccion programa | error:err | ID:err;`**:
   - Un `programa` se define como una secuencia de `instruccion` seguida de otro `programa`. Esto permite la recursividad y la capacidad de manejar múltiples instrucciones.
   - Las reglas para `error` manejan errores sintácticos, y se generan mensajes de error personalizados para ayudar en la depuración.
1. **`instruccion ::= GRAFICAR forma {: ultimaFigura = (Figura) RESULT; :} | ANIMAR OBJETO ANTERIOR animar;`**:
   - Esta regla define que una `instruccion` puede ser un comando `GRAFICAR`, que crea y agrega una nueva figura (`forma`) a la lista de objetos, o un comando `ANIMAR`, que aplica una animación a la última figura creada.

- La acción `ultimaFigura = (Figura) RESULT;` guarda la última figura creada para posibles animaciones posteriores.

1. **`forma ::= CIRCULO PAR_IZQ ID ...`**:
   - Estas reglas definen cómo se construyen las distintas figuras geométricas (círculos, cuadrados, rectángulos, líneas, polígonos) mediante sus respectivos parámetros como posición, tamaño y color.
   - Cada figura se instancia como un objeto específico (`Circulo`, `Cuadrado`, `Rectangulo`, etc.) y se agrega a la `listaObjetosList`.
1. **`color ::= AZUL | ROJO | ...`**:
   - Esta regla define los colores disponibles que pueden asignarse a las figuras. El resultado (`RESULT`) es una cadena que representa el nombre del color.
1. **`animar ::= PAR_IZQ tipoanimacion ...`**:
   - Define cómo se especifica la animación para un objeto. El tipo de animación puede ser una `curva` o una `línea`, y se pasa al método `animar(String animacion)` para realizar la animación correspondiente.
1. **`expresion ::= expresion SUMA expresion ...`**:
   - Estas reglas manejan las expresiones aritméticas dentro de las instrucciones, permitiendo sumar, restar, multiplicar, dividir, o agrupar valores numéricos.

Acciones Semánticas

Las acciones semánticas, indicadas por los bloques `{::}`, son fragmentos de código Java que se ejecutan cuando se reconoce una regla de producción específica. Estas acciones se utilizan para manipular los datos que se procesan, como crear objetos de figuras, agregar elementos a listas, o realizar cálculos.

Gramáticas Formales

```
programa ::= instruccion programa
        | error:err {: report_error("Error en el comando", err); :}
        | ID:err {: report_error("Se espera \"graficar\" o \"animar\"", err); :}
        | /*EOF*/ {: graficar(); :};

/* Instrucciones válidas */
instruccion ::= GRAFICAR forma {: ultimaFigura = (Figura) RESULT; :}
          | ANIMAR OBJETO ANTERIOR animar;

/* Formas existentes */
forma ::= CIRCULO PAR_IZQ ID:id COMA expresion:posicionX COMA expresion:posicionY
COMA expresion:radio COMA color:col PAR_DER {:
   listaObjetosList.add(new Circulo(id, (Double)posicionX, (Double)posicionY, (Double)radio,
(String)col));
:}
      | CUADRADO PAR_IZQ ID:id COMA expresion:posicionX COMA expresion:posicionY
COMA expresion:tamano COMA color:col PAR_DER {:
   listaObjetosList.add(new Cuadrado(id, (Double)posicionX, (Double)posicionY, (Double)tamano,
(String)col));
:}
      | RECTANGULO PAR_IZQ ID:id COMA expresion:posicionX COMA expresion:posicionY
COMA expresion:ancho COMA expresion:alto COMA color:col PAR_DER {:
```

```
   listaObjetosList.add(new Rectangulo(id, (Double)posicionX, (Double)posicionY,
(Double)ancho,(Double)alto, (String)col));
:}
      | LINEA_OBJ PAR_IZQ ID:id COMA expresion:inicialX COMA expresion:inicialY COMA
expresion:finalX COMA expresion:finalY COMA color:col PAR_DER {:
   listaObjetosList.add(new Linea(id, (Double)inicialX, (Double)inicialY, (Double)finalX,
(Double)finalY, (String)col));
:}
      | POLIGONO PAR_IZQ ID:id COMA expresion:posicionX COMA expresion:posicionY COMA
expresion:cantLados COMA expresion:ancho COMA expresion:alto COMA color:col PAR_DER {:
   listaObjetosList.add(new Poligono(id, (Double)posicionX, (Double)posicionY,
(Integer)((Double)cantLados).intValue(), (Double)ancho, (Double)alto, (String)col));
:}
      | error:err {: report_error("Error al graficar", err); :};

/* Colores disponibles */
color ::= AZUL    {: RESULT = "azul"; :}
      | ROJO    {: RESULT = "rojo"; :}
      | AMARILLO {: RESULT = "amarillo"; :}
      | VERDE   {: RESULT = "verde"; :}
      | NEGRO   {: RESULT = "negro"; :}
      | LILA {: RESULT = "lila"; :}
      | CAFE  {: RESULT = "cafe"; :}
      | NARANJA  {: RESULT = "naranja"; :}
      | GRIS  {: RESULT = "gris"; :};

/* Animación de objeto */
animar ::= PAR_IZQ tipoanimacion:animacion COMA expresion COMA expresion COMA expresion
PAR_DER {: animar(String.valueOf(animacion)); :};

/* Tipos de animación */
tipoanimacion ::= CURVA {: RESULT = "curva"; :}
            | LINEA {: RESULT = "línea"; :}
            | error:err {: report_error("Se esperaba curva o línea", err); :};

/* Expresiones aritméticas */
expresion ::= expresion:e1 SUMA expresion:e2              {: RESULT = ((Number)e1).doubleValue()
+ ((Number)e2).doubleValue(); :}
          | expresion:e1 RESTA expresion:e2            {: RESULT = ((Number)e1).doubleValue() -
((Number)e2).doubleValue(); :}
          | expresion:e1 MULTIPLICACION expresion:e2     {: RESULT =
((Number)e1).doubleValue() * ((Number)e2).doubleValue(); :}
          | expresion:e1 DIVISION expresion:e2          {: RESULT = ((Number)e1).doubleValue() /
((Number)e2).doubleValue(); :}
          | PAR_IZQ expresion:e PAR_DER {: RESULT = ((Number)e).doubleValue(); :}
          | NUMERO:n                                {: RESULT = ((Number)n).doubleValue(); :};
```

Paquete src/main/java/source/backend/main

1. **Importación (`import source.frontend.Principal;`)**:
   - El código importa la clase `Principal` desde el paquete `source.frontend`. Esto indica que `Principal` es la clase que contiene la interfaz gráfica (GUI) principal de la aplicación.
1. **Clase `Main`**:
   - La clase `Main` contiene el método `main`, que es el punto de entrada de la aplicación Java. Este método es lo primero que se ejecuta cuando la aplicación se inicia.
1. **Método `main`**:
   - Este método contiene la lógica para inicializar y mostrar la ventana principal de la aplicación.
   - **`System.out.println("Inicializando Clase Principal");`**:
   - Esta línea imprime un mensaje en la consola indicando que la clase `Principal` se está inicializando. Es útil para depuración y saber que la aplicación ha comenzado correctamente.
   - **`Principal inicio = new Principal();`**:
   - Se crea una instancia de la clase `Principal`, que, presumiblemente, es la ventana o interfaz principal de la aplicación.
   - **`inicio.setLocationRelativeTo(null);`**:
   - Esta línea centra la ventana de la aplicación en la pantalla. Al pasar `null` como argumento, la ventana se posiciona en el centro de la pantalla principal.
   - **`inicio.setVisible(true);`**:
   - Hace visible la ventana principal. Hasta que se llame a este método, la ventana no se mostrará en la pantalla.

Funcionalidad General

Cuando ejecutas este código:

1. Se imprime un mensaje en la consola indicando el inicio de la clase principal.
2. Se crea la ventana principal de la aplicación.
3. La ventana se centra en la pantalla.
4. Finalmente, la ventana se muestra al usuario.

Paquete src/main/java/source/backend/OpenFile

Métodos de la Clase

1. **`openFileAndSetText(JTextPane jTextPane)`**:
   - Este es el método principal de la clase que se encarga de gestionar todo el proceso de selección, apertura y carga del archivo en el `JTextPane`.
   - **Flujo del método**:
   - Se crea una instancia de `JFileChooser`, que es un cuadro de diálogo estándar para la selección de archivos.
   - Se configura un filtro de archivos para que solo muestre archivos con la extensión `.txt`.
   - Se abre el cuadro de diálogo de selección de archivos y se espera la selección del usuario.
   - Si el usuario selecciona un archivo (`JFileChooser.APPROVE_OPTION`), el archivo es leído usando el método `loadTxtFile`.
   - El contenido del archivo leído se establece en el `JTextPane` pasado como parámetro.
   - Si ocurre algún error durante la carga del archivo, se maneja mediante el método `handleFileLoadError`.
1. **`loadTxtFile(File file)`**:
   - Este método se encarga de leer el contenido de un archivo de texto y devolverlo como un `String`.

- **Flujo del método**:
- Utiliza un `FileReader` con `BufferedReader` para leer el archivo línea por línea.
- Las líneas leídas se almacenan en un `StringBuilder` y se concatenan con un salto de línea (`\n`) para mantener el formato original del archivo.
- Devuelve el contenido completo del archivo como una cadena de texto.
- Este método lanza una excepción `IOException` si ocurre algún problema durante la lectura del archivo, lo que permite que el error sea manejado por el método que lo llamó.

1. **`handleFileLoadError(Exception e)`**:
   - Este método se encarga de manejar cualquier excepción que ocurra durante la carga del archivo.
   - **Flujo del método**:
   - Imprime el stack trace de la excepción en la consola para depuración.
   - Muestra un mensaje de error en un cuadro de diálogo (`JOptionPane`) para notificar al usuario que hubo un problema al cargar el archivo. El mensaje incluye detalles sobre la excepción ocurrida.

Paquete src/main/java/source/backend/Figuras/(Cualquier Figura)

Descripción de la Clase `Cuadrado`

La clase `Cuadrado` representa un cuadrado con atributos básicos como nombre, posición en el eje X e Y, tamaño, y color. Esta clase también tiene la capacidad de "pintarse" a sí misma, es decir, de representarse gráficamente en una interfaz gráfica.

Atributos de la Clase

1. **`nombre`** (`String`):
   - Representa el nombre identificativo del cuadrado. Puede ser útil para identificar o distinguir entre diferentes figuras.
1. **`posicionX`** (`double`):
   - Representa la coordenada X en la que se ubicará la figura en el plano.
1. **`posicionY`** (`double`):
   - Representa la coordenada Y en la que se ubicará la figura en el plano.
1. **`tamano`** (`double`):
   - Define el tamaño del lado del cuadrado.
1. **`color`** (`String`):
   - Define el color del cuadrado, aunque en la implementación actual, se guarda como un `String`. Para una implementación gráfica completa, podrías considerar utilizar un objeto `Color` de AWT o similar para representar el color.

Constructor

- **`Cuadrado(String nombre, double posicionX, double posicionY, double tamano, String color)`**:
- Este constructor inicializa un objeto `Cuadrado` con los valores proporcionados para cada atributo.

Métodos Getters y Setters

- **Getters**: Métodos para obtener el valor actual de cada atributo (`getNombre`, `getPosicionX`, `getPosicionY`, `getTamano`, `getColor`).
- **Setters**: Métodos para establecer o modificar el valor de cada atributo (`setNombre`, `setPosicionX`, `setPosicionY`, `setTamano`, `setColor`).

Estos métodos siguen el patrón estándar de encapsulamiento en Java, lo que permite mantener la integridad de los datos dentro de la clase mientras se proporciona acceso controlado desde fuera de la misma.

Método `paint`

- **`paint(Graphics graphics)`**:
- Este método sobrescribe el método `paint` de la clase `Figura` (o de una clase base). En este caso, simplemente imprime un mensaje en la consola que indica que se está graficando un cuadrado (`"Grafica de Cuadrado"`).

Paquete src/main/java/source/backend/Herramientas/Numeracion

La clase `Numeracion` escucha los cambios en el caret (el cursor de texto) y en el documento del `JTextPane`, y actualiza un `JLabel` con la información de la posición actual del cursor.

Atributos de la Clase

1. **`jTextPane`** (`JTextPane`):
   - Es el componente de texto al que se le están haciendo un seguimiento para conocer la posición del caret (cursor de texto).
1. **`jLabel`** (`JLabel`):
   - Es la etiqueta que muestra la información actualizada de la posición del caret en términos de fila y columna.
1. **`lineNumber`** (`int`):
   - Almacena el número de línea actual en la que se encuentra el caret.
1. **`columnNumber`** (`int`):
   - Almacena el número de columna actual en la que se encuentra el caret.

Constructor

- **`Numeracion(JTextPane jTextPane, JLabel jLabel)`**:
- Inicializa un objeto `Numeracion` asociando el `JTextPane` y el `JLabel` proporcionados.
- Lanza una excepción si `jLabel` es `null`, ya que el componente `JLabel` es esencial para mostrar la información.
- Añade `DocumentListener` y `CaretListener` al `JTextPane` para estar atento a los cambios en el documento y la posición del caret.

Métodos Privados

- **`updateColumnNumbers()`**:
- Este método se encarga de calcular la posición actual del caret en términos de fila y columna, y actualizar el texto del `jLabel` con esta información.
- Calcula el número de línea (`caretLine`) incrementando la línea base por 1, ya que las líneas están indexadas desde 0.
- Calcula el número de columna como la diferencia entre la posición actual del caret y el inicio de la línea.

Implementación de Interfaces

1. **`CaretListener`**:
   - Implementa el método `caretUpdate(CaretEvent e)` que es llamado cada vez que el caret se mueve. Este método simplemente llama a `updateColumnNumbers()` para actualizar la información del caret.
1. **`DocumentListener`**:

- Implementa tres métodos: `insertUpdate(DocumentEvent e)`, `removeUpdate(DocumentEvent e)`, y `changedUpdate(DocumentEvent e)`. Estos métodos son llamados cuando el contenido del documento cambia, ya sea por inserción, eliminación, o cambios de atributos. En cada caso, se llama a `updateColumnNumbers()` para reflejar estos cambios.

Paquete src/main/java/source/backend/Herramientas/NumeracionFilas o NumeracionColumnas

Atributos de la Clase

1. **`component`** (`JTextComponent`):
   - El componente de texto al que se le agregará la numeración de líneas.
1. **`borderGap`** (`int`):
   - Espacio entre el borde de la numeración de líneas y el texto.
1. **`currentLineForeground`** (`Color`):
   - Color utilizado para resaltar la línea donde se encuentra el caret (cursor de texto).
1. **`minimumDisplayDigits`** (`int`):
   - Número mínimo de dígitos que se mostrarán en la columna de numeración.
1. **`lastDigits`** (`int`):
   - Número de dígitos utilizados en la última actualización, usado para recalcular el ancho de la numeración si es necesario.
1. **`lastHeight`** (`int`):
   - Altura del documento en la última actualización, usado para detectar cambios en el tamaño del texto.
1. **`lastLine`** (`int`):
   - Línea donde estaba el caret en la última actualización, usado para determinar si se debe repintar.

Constructor

- **`NumeracionFilas(JTextComponent component)`**:
- Constructor por defecto que usa un valor predeterminado de `3` para `minimumDisplayDigits`.
- **`NumeracionFilas(JTextComponent component, int minimumDisplayDigits)`**:
- Inicializa el panel con el componente de texto dado y un número mínimo de dígitos para la numeración de líneas.
- Se configura el estilo de la fuente, el espacio en el borde (`borderGap`), el color de la línea actual y el número mínimo de dígitos a mostrar.

Métodos

1. **`getBorderGap()` y `setBorderGap(int borderGap)`**:
   - Obtiene y establece el espacio entre el borde de la numeración y el texto.
   - Ajusta el ancho preferido del panel en función del nuevo espacio del borde.
1. **`getCurrentLineForeground()` y `setCurrentLineForeground(Color currentLineForeground)`**:
   - Obtiene y establece el color de la línea actual donde se encuentra el caret.
1. **`getMinimumDisplayDigits()` y `setMinimumDisplayDigits(int minimumDisplayDigits)`**:
   - Obtiene y establece el número mínimo de dígitos a mostrar en la numeración de líneas.
1. **`setPreferredWidth()`**:
   - Ajusta el ancho preferido del panel de numeración basado en el número de líneas del documento y el número mínimo de dígitos.
1. **`paintComponent(Graphics g)`**:
   - Sobrescribe el método de `JPanel` para dibujar la numeración de líneas en el margen izquierdo del componente.

- Determina si una línea es la actual (donde se encuentra el caret) y cambia el color en consecuencia antes de dibujar el número de línea.

1. **`isCurrentLine(int rowStartOffset)`**:
   - Comprueba si la línea que se está dibujando es la línea actual.
1. **`getTextLineNumber(int rowStartOffset)`**:
   - Devuelve el número de línea correspondiente a una posición de inicio de línea en el documento.
1. **`getOffsetX(int availableWidth, int stringWidth)`**:
   - Calcula la posición horizontal para alinear los números de línea a la derecha.
1. **`getOffsetY(int rowStartOffset, FontMetrics fontMetrics)`**:
   - Calcula la posición vertical del texto dentro de una línea.
1. **`caretUpdate(CaretEvent e)`**:
   - Maneja los cambios en la posición del caret. Si el caret se mueve a una nueva línea, repinta la numeración de líneas.
1. **`changedUpdate`**, **`insertUpdate`**, **`removeUpdate`**:
   - Implementa la interfaz `DocumentListener` para manejar cambios en el documento y ajustar el tamaño del panel de numeración si es necesario.
1. **`documentChanged()`**:
   - Actualiza el ancho preferido del panel de numeración y lo repinta si la altura del documento ha cambiado.
1. **`propertyChange(PropertyChangeEvent evt)`**:
   - Escucha cambios en la propiedad de la fuente del componente de texto. Si la fuente cambia, recalcula el ancho preferido.

Paquete src/main/java/source/backend/Herramientas/SaveFile

- Método saveTextToFile(JTextPane jTextPane):
- Este método muestra un cuadro de diálogo para guardar un archivo mediante JFileChooser.
- Si el usuario elige un archivo y no especifica la extensión ".txt", el programa la agrega automáticamente.
- Luego, se llama al método saveTextToFile(File file, String text) para guardar el contenido del JTextPane en el archivo seleccionado.
- Si el guardado es exitoso, se muestra un mensaje de confirmación. En caso de error, se maneja con el método handleFileSaveError(Exception e).
- Método saveTextToFile(File file, String text):
- Este método maneja la lógica de escritura en el archivo.
- Usa FileWriter y BufferedWriter para escribir el texto en el archivo especificado utilizando la codificación UTF-8.
- Se asegura de cerrar automáticamente los recursos utilizados para escribir el archivo mediante la sintaxis de try-with-resources.
- Método handleFileSaveError(Exception e):
- Este método se encarga de manejar los errores que puedan ocurrir durante el proceso de guardado.
- Muestra un cuadro de diálogo con el mensaje de error y también imprime el seguimiento de la pila (stack trace) para facilitar la depuración.

Paquete src/main/java/source/backend/Herramientas/Token

- Atributos de la clase Token:
- tipo: Almacena el tipo del token (por ejemplo, identificador, número, palabra clave, etc.).
- linea: Almacena el número de línea en la que se encuentra el token en el código fuente.

- columna: Almacena el número de columna donde comienza el token en la línea correspondiente.
- lexema: Almacena el valor del token, es decir, la cadena de caracteres que fue reconocida como un token.
- Constructor:
- El constructor toma cuatro parámetros (tipo, lexema, linea, columna) y los asigna a los atributos correspondientes. Esto permite la creación de objetos Token con toda la información necesaria para representar un token del código fuente.
- Métodos get y set:
- Estos métodos permiten acceder y modificar los valores de los atributos tipo, linea, columna, y lexema.
- Los métodos get retornan el valor actual del atributo, mientras que los métodos set permiten cambiar dicho valor.
- Método toString:
- El método toString sobrescrito devuelve una representación en cadena del objeto Token, lo cual es útil para depuración o impresión de información del token. Incluye el tipo, la línea, la columna y el lexema del token.

Paquete src/main/java/source/backend/Reportes/Errores

- Atributos de la clase Errores:
- lexema: Almacena la cadena de caracteres que causó el error.
- linea: Almacena el número de línea en el código fuente donde se detectó el error.
- columna: Almacena la columna en la línea donde comienza el error.
- tipo: Almacena el tipo de error (por ejemplo, léxico, sintáctico, semántico, etc.).
- descripcion: Almacena una descripción más detallada del error, explicando qué fue lo que falló.
- Constructor:
- El constructor de la clase toma cinco parámetros (lexema, linea, columna, tipo, descripcion) y los asigna a los atributos correspondientes. Esto permite crear instancias de Errores con toda la información relevante sobre un error específico.
- Métodos get y set:
- Estos métodos permiten acceder y modificar los valores de los atributos de la clase:
- getLexema y setLexema para obtener y modificar el lexema.
- getLinea y setLinea para obtener y modificar la línea.
- getColumna y setColumna para obtener y modificar la columna.
- getTipo y setTipo para obtener y modificar el tipo de error.
- getDescripcion y setDescripcion para obtener y modificar la descripción del error.
- Método toString:
- El método toString sobrescrito devuelve una representación en cadena de un objeto Errores. Esta representación incluye el lexema, la línea, la columna, el tipo de error y la descripción del error. Es útil para la depuración o para registrar los errores en un informe de errores.

Paquete src/main/java/source/frontend/Principal

Resumen de la Clase `Principal`

1. **Atributos de la Clase**:
   - `panelBlanco`: Un `JTextPane` para editar texto.
   - `imagenFondo`: Imagen de fondo para la ventana.
   - `relojLabel`: Etiqueta para mostrar la hora.

- `contenedorPanel`: Panel principal para el contenido.
- `relojActivo`: Booleano para activar o desactivar el reloj.
- `tamañoPanelFondo`: Dimensiones del panel de fondo.
- `titulo`: Título de la ventana.
- `numerosFilaC1`: Objeto para numerar las filas del `JTextPane`.
- `jScrollPane1`: Panel de desplazamiento para `panelBlanco`.
- `numeracion`: Objeto para manejar la numeración y la información de filas y columnas.
- `infoLabel`: Etiqueta que muestra la información de fila y columna actuales.
- `labelInferiorIzquierda`: Etiqueta en la esquina inferior izquierda.
- `ocurrenciaOperadoresMatematicos`: Objeto para manejar las ocurrencias de operadores matemáticos.

1. **Constructor `Principal`**:
   - Inicializa la interfaz de usuario y el reloj.
   - Configura el `JTextPane`, el menú, el reloj y los botones.
1. **Métodos Principales**:
   - `initUI()`: Configura la interfaz de usuario principal, incluyendo el `JTextPane`, los botones y el menú.
   - `addMenus(JMenuBar menuPrincipal)`: Añade menús y elementos al `JMenuBar`.
   - `createScaledIcon(String path, int width, int height)`: Crea y escala iconos para los elementos del menú.
   - `addMessageLabel(JMenuBar menuPrincipal)`: Añade una etiqueta con un mensaje al `JMenuBar`.
   - `addClockLabel(JMenuBar menuPrincipal)`: Añade una etiqueta con la hora al `JMenuBar`.
   - `iniciarReloj()`: Inicia un hilo para actualizar el reloj cada segundo.
   - `actualizarReloj()`: Actualiza la etiqueta del reloj.
   - `actualizarInfoLabel(int fila, int columna)`: Actualiza la etiqueta con la información de la fila y columna actuales.
1. **Método `main`**:
   - Configura el aspecto del `JFrame` y muestra la ventana.

Mejoras y Comentarios

1. **Estilo de Código**:
   - El código está bien organizado, pero podrías considerar extraer partes del código de inicialización del menú en métodos separados para mayor claridad.
1. **Gestión del Reloj**:
   - El método `iniciarReloj()` crea un nuevo hilo para actualizar el reloj. Podrías considerar usar un `javax.swing.Timer` para simplificar la actualización del reloj en el hilo del evento de despacho.
1. **Manejo de Excepciones**:
   - Cuando manejas excepciones en el botón de compilar, es una buena práctica proporcionar mensajes de error más detallados si es posible, o permitir que el usuario sepa qué debe hacer a continuación.
1. **Mejora de Usabilidad**:
   - Podrías proporcionar más retroalimentación al usuario cuando se realizan acciones como guardar o abrir archivos, como mostrar mensajes de confirmación.
1. **Optimización del Reloj**:
   - La forma en que manejas el reloj podría mejorarse usando `java.util.Timer` o `javax.swing.Timer` para actualizar la interfaz gráfica, en lugar de crear un hilo manualmente.

1. **Documentación**:
   - Añadir más comentarios y documentación puede ayudar a otros desarrolladores (o a ti mismo en el futuro) a entender el propósito de cada método y atributo.

Paquete src/main/java/source/frontend/PanelGraficas

`PanelGraficas` es una clase que extiende `JPanel` y se encarga de mostrar gráficos de diferentes figuras en un panel de interfaz gráfica de usuario. Además, permite la exportación de estos gráficos en dos formatos: PNG y PDF. La clase utiliza diversas herramientas para renderizar las figuras y gestionar la exportación.

Atributos

- **`COLOR_ROJO`**: Color para figuras rojas.
- **`COLOR_AZUL`**: Color para figuras azules.
- **`COLOR_VERDE`**: Color para figuras verdes.
- **`COLOR_NEGRO`**: Color para figuras negras.
- **`COLOR_AMARILLO`**: Color para figuras amarillas.
- **`COLOR_LILA`**: Color para figuras lilas.
- **`COLOR_CAFE`**: Color para figuras café.
- **`COLOR_GRIS`**: Color para figuras grises.
- **`COLOR_NARANJA`**: Color para figuras naranjas.
- **`listaFiguras`**: Lista de objetos `Figura` que contiene las figuras a dibujar en el panel.
- **`btnExportar`**: Botón que permite al usuario exportar el contenido del panel en diferentes formatos.

Constructor

`PanelGraficas()`

Inicializa el panel con un diseño `BorderLayout`, establece el fondo en blanco, y agrega un borde negro de 2 píxeles. También crea un botón "Exportar" que, al hacer clic, muestra un menú para seleccionar el formato de exportación.

Métodos

- **`public List<Figura> getListaFiguras()`**
- Devuelve la lista de figuras actualmente configuradas en el panel.
- **`public void setListaFiguras(List<Figura> listaFiguras)`**
- Establece la lista de figuras a dibujar en el panel y solicita una repintura del panel para reflejar los cambios.
- **`protected void paintComponent(Graphics g)`**
- Sobrescribe el método `paintComponent` de `JPanel`. Utiliza `Graphics2D` para dibujar las figuras en el panel. Configura el renderizado para una calidad óptima y llama a métodos específicos para dibujar cada tipo de figura basado en su instancia (`Circulo`, `Linea`, `Cuadrado`, `Rectangulo`, `Poligono`).
- **`private void dibujarCirculo(Graphics2D g2d, Circulo circulo)`**
- Dibuja un círculo utilizando el objeto `Circulo`. Configura el color basado en el valor proporcionado por `devolverColor`.
- **`private void dibujarLinea(Graphics2D g2d, Linea linea)`**
- Dibuja una línea utilizando el objeto `Linea`. Configura el color y el grosor de la línea.
- **`private void dibujarCuadrado(Graphics2D g2d, Cuadrado cuadrado)`**
- Dibuja un cuadrado utilizando el objeto `Cuadrado`. Configura el color y el tamaño del cuadrado.
- **`private void dibujarRectangulo(Graphics2D g2d, Rectangulo rectangulo)`**

- Dibuja un rectángulo utilizando el objeto `Rectangulo`. Configura el color y las dimensiones del rectángulo.
- **`private void dibujarPoligono(Graphics2D g2d, Poligono poligono)`**
- Dibuja un polígono utilizando el objeto `Poligono`. Calcula los puntos del polígono basándose en el número de lados y las dimensiones proporcionadas.
- **`private Color devolverColor(String color)`**
- Devuelve un objeto `Color` basado en el nombre de color proporcionado como una cadena. Si el color no está en la lista predefinida, devuelve `Color.WHITE`.
- **`public void guardarComoPNG()`**
- Abre un cuadro de diálogo para que el usuario seleccione la ubicación y el nombre del archivo para guardar el gráfico en formato PNG. Crea una imagen del panel y la guarda en el archivo especificado.
- **`public void guardarComoPDF()`**
- Abre un cuadro de diálogo para que el usuario seleccione la ubicación y el nombre del archivo para guardar el gráfico en formato PDF. Crea una imagen del panel y la guarda en un archivo PDF utilizando `iText`.
- **`private byte[] imageToByteArray(BufferedImage image) throws IOException`**
- Convierte una imagen `BufferedImage` en un array de bytes. Utiliza un `ByteArrayOutputStream` para escribir la imagen en formato PNG y devuelve los bytes correspondientes.

Paquete src/main/java/source/frontend/Graficar/(cualquier metodo de graficar)

`GrafCirculo` es una clase que extiende `JPanel` y está diseñada para representar un círculo en un panel de interfaz gráfica de usuario. Esta clase gestiona las propiedades del círculo, como la posición, el radio y el color, y se encarga de dibujarlo en el panel.

Atributos

- **`posicionX`**: Coordenada X de la posición del centro del círculo.
- **`posicionY`**: Coordenada Y de la posición del centro del círculo.
- **`radio`**: Radio del círculo.
- **`color`**: Color del círculo, especificado como una cadena de texto.

Constructor

`GrafCirculo()`

Inicializa una instancia de `GrafCirculo`. Imprime un mensaje en la consola para confirmar la creación del objeto.

Métodos

- **`public double getPosicionX()`**
- Devuelve la coordenada X del centro del círculo.
- **`public void setPosicionX(double posicionX)`**
- Establece la coordenada X del centro del círculo.
- **`public double getPosicionY()`**
- Devuelve la coordenada Y del centro del círculo.
- **`public void setPosicionY(double posicionY)`**
- Establece la coordenada Y del centro del círculo.
- **`public double getRadio()`**
- Devuelve el radio del círculo.
- **`public void setRadio(double radio)`**

- Establece el radio del círculo.
- **`public String getColor()`**
- Devuelve el color del círculo como una cadena de texto.
- **`public void setColor(String color)`**
- Establece el color del círculo a partir de una cadena de texto.
- **`@Override public void paintComponent(Graphics g)`**
- Sobrescribe el método `paintComponent` de `JPanel`. Este método se encarga de dibujar el círculo en el panel.
- Configura el color del gráfico (`Graphics`) según el valor de la propiedad `color`.
- Dibuja un círculo usando el método `fillOval`, el cual se ajusta según la posición y el radio del círculo.
- Utiliza un `switch` para seleccionar el color correcto basado en el valor de `color`. Si el valor no coincide con ninguno de los casos predefinidos, se establece el color como blanco (`Color.WHITE`).

Paquete src/main/java/source/frontend/PanelReporte/OcurrenciaOperadoresMatematicos

`OcurrenciaOperadoresMatematicos` es una clase que extiende `JPanel` y está diseñada para mostrar una tabla con la ocurrencia de operadores matemáticos en una interfaz gráfica de usuario. Esta clase gestiona la visualización de datos sobre operadores matemáticos, permite la actualización de la tabla y ofrece funciones para filtrar y mostrar los datos.

Atributos

- **`operadorData`**: Lista de listas que contiene los datos de los operadores matemáticos. Cada lista interna representa una fila en la tabla.
- **`model`**: Modelo de tabla (`DefaultTableModel`) utilizado para gestionar los datos de la tabla.
- **`sorter`**: `TableRowSorter` que permite ordenar y filtrar las filas de la tabla.
- **`jScrollPane1`**: Componente `JScrollPane` que proporciona una barra de desplazamiento para la tabla.
- **`jTable1`**: Componente `JTable` que muestra los datos de los operadores matemáticos.

Constructor

`public OcurrenciaOperadoresMatematicos()`

Inicializa una instancia de `OcurrenciaOperadoresMatematicos`. Configura el modelo de la tabla y el `TableRowSorter` para permitir el filtrado y la ordenación de las filas. Llama al método `initComponents()` para inicializar los componentes gráficos.

Métodos

- **`private void initComponents()`**
- Inicializa y configura los componentes gráficos del panel:
- Crea y configura `jScrollPane1` y `jTable1`.
- Establece el modelo de tabla con las columnas "Operador" y "Cantidad", y especifica que las celdas no son editables.
- Configura el layout del panel y agrega el `JScrollPane` que contiene la tabla.
- **`public void actualizarTabla(List<List<Object>> operadorList)`**
- Actualiza los datos de la tabla:
- Borra las filas existentes en la tabla.
- Agrega las nuevas filas de datos proporcionadas en `operadorList`.
- Actualiza el `TableRowSorter` con el nuevo modelo de tabla.

- **`private void filtrarTablaPorOperador(String operador)`**
- Filtra las filas de la tabla para mostrar solo aquellas que coincidan con el operador proporcionado:
- Si el parámetro `operador` está vacío, se elimina el filtro y se muestran todas las filas.
- Si `operador` tiene un valor, se aplica un filtro de expresión regular para mostrar solo las filas que contienen el operador especificado en la primera columna.
- **`private void mostrarContenidoOriginal()`**
- Muestra todos los datos en la tabla eliminando cualquier filtro aplicado previamente.

Paquete src/main/jflex

El archivo JFlex define un analizador léxico para un lenguaje específico, utilizando expresiones regulares para identificar diferentes tokens en el texto de entrada. A continuación se detalla el contenido del archivo:

Directivas

- **`%public`**: Indica que la clase generada será pública.
- **`%class Lexer`**: Define el nombre de la clase generada como `Lexer`.
- **`%unicode`**: Permite el manejo de caracteres Unicode.
- **`%line`**: Habilita el seguimiento de números de línea en el texto de entrada.
- **`%column`**: Habilita el seguimiento de números de columna en el texto de entrada.
- **`%ignorecase`**: Hace que el análisis sea insensible a mayúsculas y minúsculas.
- **`%cup`**: Indica que el analizador léxico se integrará con CUP (Constructor de Analizadores de Parseo).

Expresiones Regulares

Las expresiones regulares definen patrones para identificar distintos tipos de tokens en el texto. Cada expresión regular está asociada con una acción que crea un `Symbol` (token) correspondiente. Aquí se explica cada una:

- **`ID = [a-zA-Z_][a-zA-Z0-9_]*[a-zA-Z0-9-]*`**:
- Define un identificador como una cadena que comienza con una letra o guion bajo, seguida por una combinación de letras, números, guiones bajos o guiones.

Código de Acción

- **`"azul" { return new Symbol(sym.AZUL, yyline+1, yycolumn+1, yytext()); }`**
- Reconoce el literal `"azul"` y lo convierte en un `Symbol` con el valor `sym.AZUL`. La línea y columna se ajustan para ser 1-basadas, y `yytext()` proporciona el texto del token.
- **`"línea" { return new Symbol(sym.LINEA, yyline+1, yycolumn+1, yytext()); }`**
- Reconoce el literal `"línea"` y lo convierte en un `Symbol` con el valor `sym.LINEA`.
- **`[ \t\n\r\f] {}`**
- Ignora los espacios en blanco y los caracteres de nueva línea.
- **`"+" { operadorAritmetico.add(new OperadorAritmetico("SUMA", yytext(), yyline+1, yycolumn+1, 1)); return new Symbol(sym.SUMA, yyline+1, yycolumn+1, yytext()); }`**
- Reconoce el símbolo `"+"`, lo agrega a una lista de `OperadorAritmetico` y lo convierte en un `Symbol` con el valor `sym.SUMA`.
- **`{ID} { return new Symbol(sym.ID, yytext()); }`**
- Reconoce identificadores y los convierte en un `Symbol` con el valor `sym.ID`.
- **`[0-9]+\.[0-9]+ { return new Symbol(sym.NUMERO, Double.parseDouble(yytext())); }`**
- Reconoce números decimales y los convierte en un `Symbol` con el valor `sym.NUMERO`.
- **`[0-9]+ { return new Symbol(sym.NUMERO, Integer.parseInt(yytext())); }`**
- Reconoce números enteros y los convierte en un `Symbol` con el valor `sym.NUMERO`.

- **`<<EOF>> { return new Symbol(sym.EOF); }`**
- Indica el final del archivo y devuelve un `Symbol` con el valor `sym.EOF`.
- **`[^] { errores.add(new Errores(yytext(), yyline + 1, yycolumn + 1, "Léxico", "Caracter desconocido: " + yytext())); System.err.println("Error léxico: " + yytext() + " linea: " + String.valueOf(yyline + 1) + " columna: " + String.valueOf(yycolumn + 1)); }`**
- Maneja caracteres desconocidos que no coinciden con ninguna de las expresiones regulares anteriores. Los agrega a una lista de errores y se imprime un mensaje de error en la salida estándar de error.

Cláusulas Adicionales

- **`%{ ... %}`**: Bloque de código Java que se incluye directamente en la clase generada:
- Se definen tres listas estáticas (`errores`, `tokens`, `operadorAritmetico`) para almacenar errores léxicos, tokens generados y operadores aritméticos encontrados, respectivamente.
- Se proporciona un método `getErrores()` para acceder a la lista de errores.
- **`%state OPERATOR_STATE`**: Declara un estado adicional `OPERATOR_STATE` para el analizador léxico, aunque no se utiliza en las expresiones regulares proporcionadas.

Cuando se utiliza JFlex y CUP para generar un analizador léxico y un analizador sintáctico en Java, se generan varios archivos que desempeñan roles específicos en el proceso de análisis. A continuación, se detalla la función de los archivos generados por JFlex (`Lexer`), CUP (`Parser` y `sym`), y cómo interactúan entre sí:

1. Archivo `sym.java`

- **Propósito**: Define los símbolos terminales y no terminales utilizados por el analizador sintáctico generado por CUP. Contiene constantes que representan los distintos tipos de tokens que el analizador léxico puede devolver y que el analizador sintáctico procesará.
- **Contenido**:
- **Símbolos Terminales**: Representan los tokens que se extraen del texto de entrada. Cada símbolo terminal se define como una constante estática final con un valor entero único. Ejemplos: `sym.ID`, `sym.NUMERO`, `sym.SUMA`, etc.
- **Símbolos No Terminales**: Se utilizan en el archivo CUP para definir los diferentes tipos de construcciones sintácticas que el parser reconocerá. Estos símbolos suelen tener nombres más descriptivos y son utilizados en las reglas de producción.

2. Archivo `Lexer.java`

- **Propósito**: Generado por JFlex, es la clase del analizador léxico que procesa el texto de entrada y genera tokens que el analizador sintáctico utilizará.
- **Contenido**:
- **Métodos**: Incluye métodos para inicializar el analizador, leer el texto y devolver tokens. El método principal es `next_token()`, que analiza el texto y devuelve un `Symbol` correspondiente al token encontrado.
- **Expresiones Regulares**: Define cómo se deben identificar los diferentes tokens usando patrones de expresiones regulares.
- **Acciones**: Las acciones asociadas con las expresiones regulares crean instancias de `Symbol` con el tipo de token, la línea y la columna.

3. Archivo `Parser.java`

- **Propósito**: Generado por CUP, es la clase del analizador sintáctico que utiliza los tokens generados por el analizador léxico para construir una estructura sintáctica (árbol de análisis) basada en las reglas de producción definidas en el archivo CUP.
- **Contenido**:
- **Métodos**: Incluye el método `parse()` que inicia el proceso de análisis sintáctico.
- **Reglas de Producción**: Define cómo se deben combinar los tokens para formar construcciones válidas del lenguaje. Estas reglas utilizan símbolos terminales y no terminales.

Interacción entre `Lexer`, `Parser` y `sym`

1. **Análisis Léxico (`Lexer`)**:
   - El archivo JFlex (`Lexer.java`) analiza el texto de entrada y genera tokens. Cada token es una instancia de `Symbol`, que incluye un tipo (definido en `sym.java`), la línea y la columna del token en el texto de entrada.
1. **Análisis Sintáctico (`Parser`)**:
   - El archivo CUP (`Parser.java`) recibe los tokens generados por el `Lexer` y los utiliza para construir una estructura sintáctica basada en las reglas definidas en el archivo CUP. Utiliza los símbolos terminales definidos en `sym.java` para identificar y procesar los tokens.
1. **Símbolos (`sym.java`)**:
   - Define los valores constantes que representan los diferentes tipos de tokens y construcciones en el lenguaje. Estos símbolos son utilizados tanto por el `Lexer` como por el `Parser` para referirse a los distintos tipos de tokens y no terminales.

Resumen

- **`sym.java`**: Define las constantes para los símbolos terminales y no terminales.
- **`Lexer.java`**: Analiza el texto de entrada y genera tokens.
- **`Parser.java`**: Construye una estructura sintáctica a partir de los tokens generados por el `Lexer`.