Universidad San Carlos de Guatemala

Facultad de Ingeniería

Organización De Lenguajes Y Compiladores, Sección C

Ing. Kevin Adiel Lajpop Ajpacaja

Auxiliar: Carlos Daniel Acabal Pérez

Primer Semestre 2024

Tema:

Manuel Técnico – Compi Script

Nombre: Carlos Manuel Lima y Lima

Registro Académico: 202201524

CUI: 3009368850101

Guatemala, 21 de abril del 2024.

Clase Tipo

La clase Tipo es una estructura de datos que representa un tipo de dato en un lenguaje de programación o sistema. Esta clase tiene un atributo tipo que puede ser uno de los valores definidos en el enum tipo_dato. La clase proporciona métodos para establecer y obtener el valor de tipo, y un método para obtener el nombre del tipo como una cadena de texto.

Enum tipo_dato: Este es un tipo especial en TypeScript que permite definir un conjunto de constantes nombradas. En este caso, se definen seis tipos de datos: ENTERO, DECIMAL, BOOLEANO, CARACTER, CADENA, VOID.

Clase Tipo: Esta clase tiene un atributo privado llamado tipo que puede ser uno de los valores del enum tipo_dato.

Constructor: El constructor de la clase Tipo acepta un argumento tipo que es un valor del enum tipo_dato. Este valor se asigna al atributo tipo de la clase.

Método setTipo: Este método acepta un argumento tipo que es un valor del enum tipo_dato y lo asigna al atributo tipo de la clase.

Método getTipo: Este método devuelve el valor actual del atributo tipo.

Método getNombreTipo: Este método devuelve el nombre del tipo de dato como una cadena de texto. Utiliza una declaración switch para determinar qué cadena de texto devolver en función del valor del atributo tipo.

Clase Símbolo

La clase Símbolo representa un símbolo en un lenguaje de programación o sistema. Cada símbolo tiene un tipo (que es una instancia de la clase Tipo), un id que es una cadena de texto, un valor que puede ser cualquier tipo de dato, y las coordenadas fila y columna que representan la ubicación del símbolo en el código fuente.

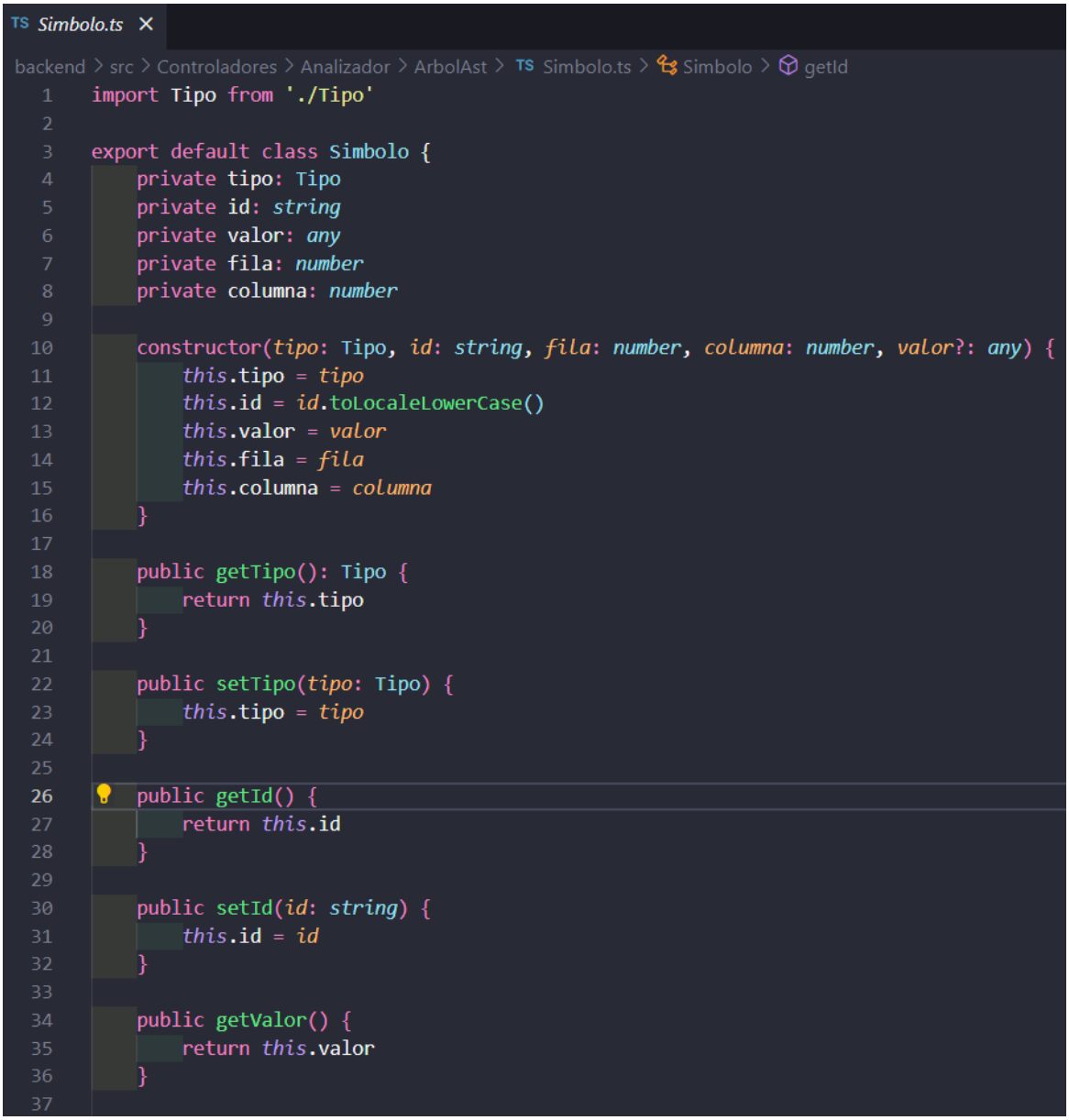
Importación de la clase Tipo: La clase Tipo se importa desde otro archivo. Esta clase se utiliza para definir el tipo del símbolo.

Clase Símbolo: Esta clase tiene cinco atributos privados: tipo, id, valor, fila y columna.

Constructor: El constructor de la clase Símbolo acepta cinco argumentos: tipo, id, fila, columna y valor. El argumento valor es opcional. El id se convierte a minúsculas antes de ser asignado al atributo id.

Métodos get y set: Estos métodos permiten obtener y establecer los valores de los atributos de la clase. Los métodos get devuelven el valor del atributo correspondiente, mientras que los métodos set asignan un nuevo valor al atributo correspondiente.

Métodos getFila y getColumna: Estos métodos devuelven los valores de los atributos fila y columna, respectivamente. Estos atributos representan la ubicación del símbolo en el código fuente.



Clase Tabla Símbolo

La clase TablaSimbolo representa una tabla de símbolos, que es una estructura de datos utilizada por los compiladores para guardar información sobre identificadores (variables, funciones, clases, etc.) definidos en el código. Cada entrada en la tabla es un objeto de las clases Simbolo, SimboloA o SimboloM.

Importación de las clases Simbolo, SimboloA y SimboloM: Estas clases se importan desde otros archivos. Se utilizan para definir los tipos de los símbolos que se pueden almacenar en la tabla.

Clase TablaSimbolo: Esta clase tiene tres atributos privados: tabla_anterior, tabla_actual y nombre. tabla_anterior es una referencia a la tabla de símbolos del ámbito anterior (si existe), tabla_actual es un mapa que almacena los símbolos del ámbito actual y nombre es una cadena de texto que representa el nombre de la tabla.

Constructor: El constructor de la clase TablaSimbolo acepta un argumento opcional anterior que es una referencia a la tabla de símbolos del ámbito anterior. Inicializa tabla_actual como un nuevo mapa y nombre como una cadena de texto vacía.

Métodos get y set: Estos métodos permiten obtener y establecer los valores de los atributos de la clase. Los métodos get devuelven el valor del atributo correspondiente, mientras que los métodos set asignan un nuevo valor al atributo correspondiente.

Métodos getMatriz, setMatriz, getArreglo, setArreglo, getVariable y setVariable: Estos métodos permiten obtener y establecer los valores de los símbolos de tipo matriz, arreglo y variable, respectivamente. Los métodos get buscan un símbolo con un determinado identificador en la tabla actual y en las tablas anteriores. Los métodos set añaden un nuevo símbolo a la tabla actual si no existe un símbolo con el mismo identificador.

Métodos getNombre y setNombre: Estos métodos permiten obtener y establecer el valor del atributo nombre.

TablaSimbolo.ts

backend > src > Controladores > Analizador > ArbolAst > TablaSimbolo.ts > TablaSimbolo > setAnterior

```ts
import Simbolo from "./Simbolo";
import SimboloA from "./SimboloA";
import SimboloM from "./SimboloM";

export default class TablaSimbolo {
    private tabla_anterior: TablaSimbolo | any
    private tabla_actual: Map<string, SimboloM|SimboloA|Simbolo>
    private nombre: string

    constructor(anterior?: TablaSimbolo) {
        this.tabla_anterior = anterior
        this.tabla_actual = new Map<string, SimboloM|SimboloA|Simbolo>()
        this.nombre = ""
    }
```

Clase Tabla Arbol

La clase Arbol representa un árbol de análisis, que es una estructura de datos utilizada por los compiladores para representar el código fuente de un programa. Cada nodo del árbol es una instrucción y el árbol completo representa la estructura del programa.

Importación de módulos y clases: Se importan varios módulos y clases necesarios para la definición de la clase Arbol. Estos incluyen el módulo fs de Node.js para operaciones de archivos, las clases TablaSimbolo, Instruccion, Errores y Metodo.

Clase Arbol: Esta clase tiene seis atributos privados: instrucciones, consola, tabla_global, errores, funciones y lista_tablas.

Constructor: El constructor de la clase Arbol acepta un argumento instrucciones que es un array de instrucciones. Inicializa consola como una cadena de texto vacía, tabla_global como una nueva tabla de símbolos, errores y funciones como arrays vacíos y lista_tablas como un array vacío.

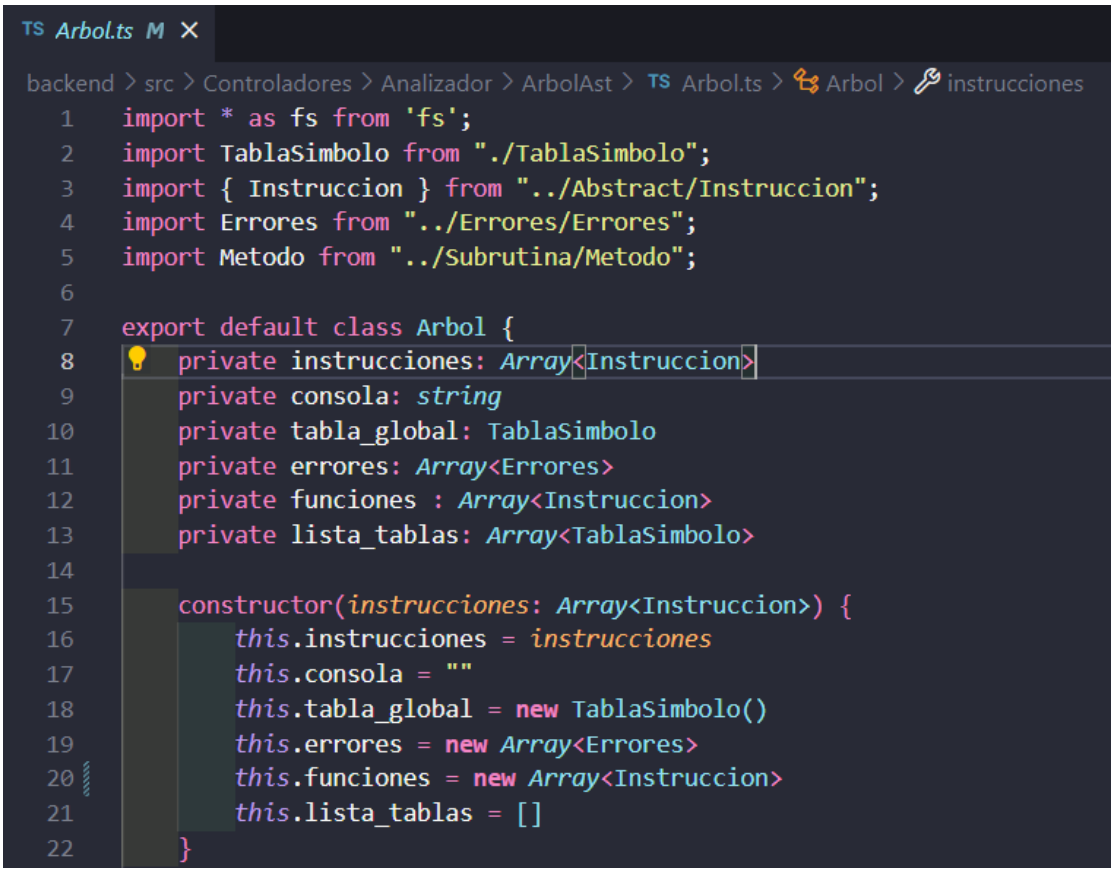
Métodos get y set: Estos métodos permiten obtener y establecer los valores de los atributos de la clase. Los métodos get devuelven el valor del atributo correspondiente, mientras que los métodos set asignan un nuevo valor al atributo correspondiente.

Métodos Cout y CoutEndl: Estos métodos permiten agregar contenido a la consola. Cout agrega el contenido sin un salto de línea al final, mientras que CoutEndl agrega el contenido con un salto de línea al final.

Método agregarTabla: Este método permite agregar una nueva tabla de símbolos a la lista de tablas.

Método agregarError: Este método permite agregar un nuevo error a la lista de errores.

Métodos addFunciones y getFuncion: Estos métodos permiten agregar una nueva función a la lista de funciones y obtener una función de la lista de funciones por su identificador, respectivamente.



Clase Instrucción

La clase Instruccion es una clase abstracta que representa una instrucción genérica en un lenguaje de programación o sistema. Cada instrucción tiene un tipo_dato, y las coordenadas fila y columna que representan la ubicación de la instrucción en el código fuente.

Importación de las clases Arbol, TablaSimbolos y Tipo: Estas clases se importan desde otros archivos. Se utilizan para definir los tipos de los atributos de la clase Instruccion.

Clase abstracta Instruccion: Esta es una clase abstracta, lo que significa que no se pueden crear instancias de esta clase directamente. En su lugar, esta clase debe ser extendida por otras clases. La clase Instruccion tiene tres atributos públicos: tipo_dato, fila y columna.

Constructor: El constructor de la clase Instruccion acepta tres argumentos: tipo, fila y columna. Estos valores se asignan a los atributos correspondientes de la clase.

Método abstracto interpretar: Este es un método abstracto, lo que significa que las clases que extienden la clase Instruccion deben proporcionar una implementación para este método. El método interpretar acepta dos argumentos: arbol y tabla, que son instancias de las clases Arbol y TablaSimbolos, respectivamente. El método interpretar debe devolver un valor, pero el tipo de este valor no está especificado (es any).

TS *Instruccion.ts* ×

backend > src > Controladores > Analizador > Abstract > TS Instruccion.ts > Instruccion

```ts
import Arbol from "../ArbolAst/Arbol";
import TablaSimbolos from "../ArbolAst/TablaSimbolo";
import Tipo from "../ArbolAst/Tipo";

export abstract class Instruccion {
    public tipo_dato: Tipo
    public fila: number
    public columna: number

    constructor(tipo: Tipo, fila: number, columna: number) {
        this.tipo_dato = tipo
        this.fila = fila
        this.columna = columna
    }

    abstract interpretar(arbol: Arbol, tabla: TablaSimbolos): any

}
```

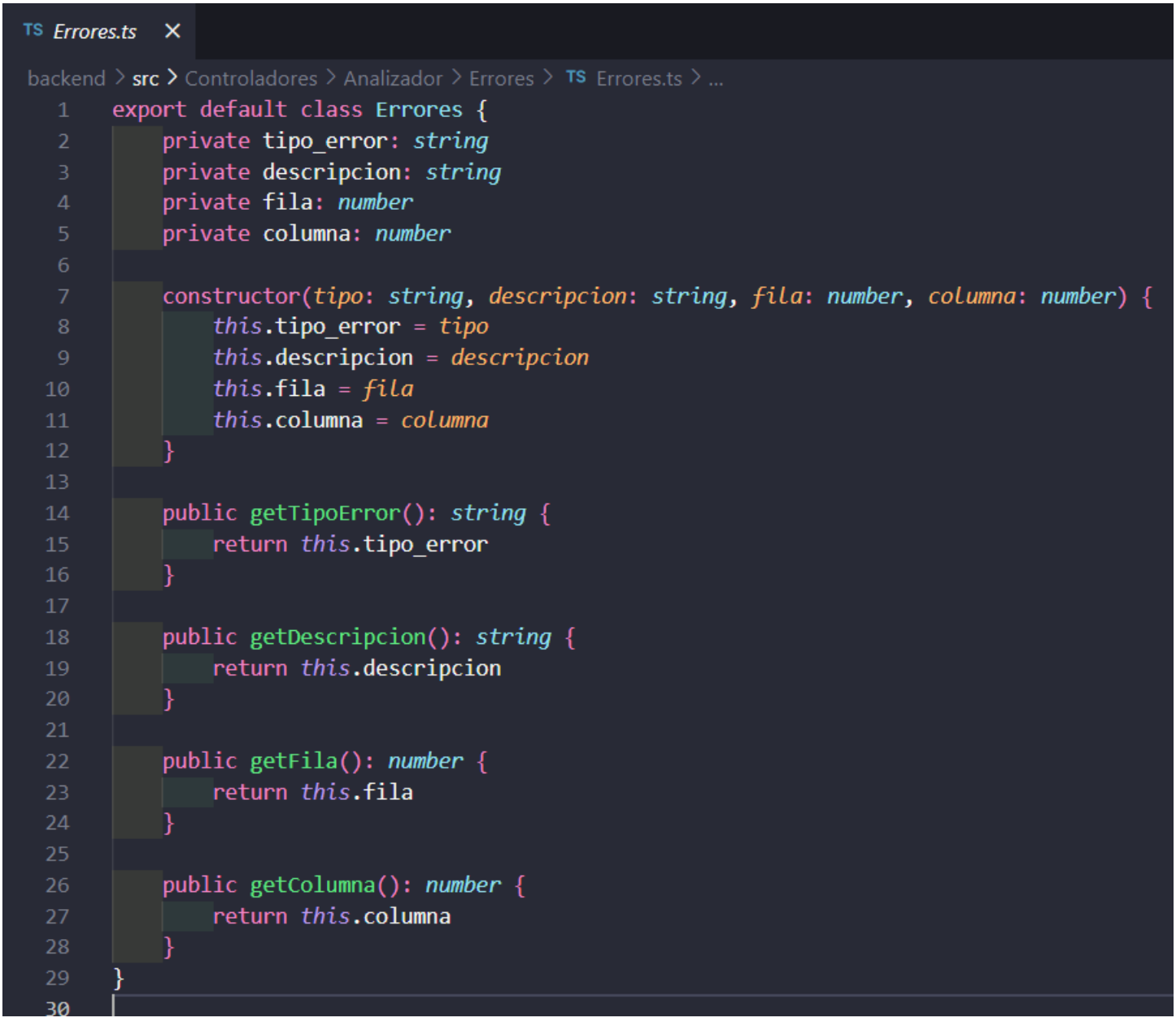
Clases Errores

La clase Errores representa un error en un lenguaje de programación o sistema. Cada error tiene un tipo_error, una descripcion, y las coordenadas fila y columna que representan la ubicación del error en el código fuente.

Clase Errores: Esta clase tiene cuatro atributos privados: tipo_error, descripcion, fila y columna.

Constructor: El constructor de la clase Errores acepta cuatro argumentos: tipo, descripcion, fila y columna. Estos valores se asignan a los atributos correspondientes de la clase.

Métodos get: Estos métodos permiten obtener los valores de los atributos de la clase. Los métodos getTipoError, getDescripcion, getFila y getColumna devuelven el valor del atributo correspondiente. Estos métodos son útiles para obtener información detallada sobre el error después de que se ha creado una instancia de la clase Errores.



Clase Asignación

La clase Asignacion es una subclase de la clase abstracta Instruccion que representa una instrucción de asignación en un lenguaje de programación. Esta instrucción asigna un valor a una variable.

Clase Asignacion: Esta clase extiende la clase abstracta Instruccion. Tiene dos atributos privados: Identificador, que es el nombre de la variable a la que se va a asignar un valor, y expresion, que es la expresión cuyo valor se va a asignar a la variable.

Constructor: El constructor de la clase Asignacion acepta cuatro argumentos: Identificador, expresion, fila y columna. Estos valores se asignan a los atributos correspondientes de la clase.

Método interpretar: Este método implementa el método abstracto interpretar de la clase Instruccion. Primero, interpreta la expresión y obtiene su valor. Si el valor es una instancia de la clase Errores, lo devuelve inmediatamente. Luego, busca la variable en la tabla de símbolos. Si la variable no existe, crea un nuevo error, lo agrega a la lista de errores del árbol y lo devuelve. Si la variable existe pero su tipo no coincide con el tipo de la expresión, crea un nuevo error, lo agrega a la lista de errores del árbol y lo devuelve. Finalmente, si no hay errores, asigna el nuevo valor a la variable y actualiza el tipo de la instrucción de asignación al tipo de la variable.

TS Asignacion.ts

backend > src > Controladores > Analizador > Instrucciones > TS Asignacion.ts > Asignacion

```ts
import { Instruccion } from "../Abstract/Instruccion";
import Errores from "../Errores/Errores";
import Arbol from "../ArbolAst/Arbol";
import TablaSimbolo from "../ArbolAst/TablaSimbolo";
import Tipo, { tipo_dato } from '../ArbolAst/Tipo'

export default class Asignacion extends Instruccion {
    private Identificador: string
    private expresion: Instruccion

    constructor(Identificador: string, expresion: Instruccion, fila: number, columna: number) {
        super(new Tipo(tipo_dato.VOID), fila, columna)
        this.Identificador = Identificador
        this.expresion = expresion
    }

    interpretar(arbol: Arbol, tabla: TablaSimbolo) {
        let nuevo_valor = this.expresion.interpretar(arbol, tabla)
        if (nuevo_valor instanceof Errores) return nuevo_valor
        let valor = tabla.getVariable(this.Identificador.toLocaleLowerCase())
        if (valor == null){
            let error = new Errores("Semántico", "Variable No Existente", this.fila, this.columna)
            arbol.agregarError(error);
            arbol.setConsola("Semántico: Variable No Existente.\n")
            return error
        }
        if (this.expresion.tipo_dato.getTipo() != valor.getTipo().getTipo()){
            let error = new Errores("Semántico", "Asignación Incorrecta", this.fila, this.columna)
            arbol.agregarError(error);
            arbol.setConsola("Semántico: Asignación Incorrecta.\n")
            return error
        }
        this.tipo_dato = valor.getTipo()
        valor.setValor(nuevo_valor)
    }
}
```

Abstracción De Instrucciones

La abstracción en la programación orientada a objetos es un proceso que implica la creación de clases simples que representan la funcionalidad de sistemas más complejos. En el caso de las clases Instruccion, Asignacion, Errores, Arbol, TablaSimbolo y Tipo, se utilizan varios niveles de abstracción para representar las diferentes partes de un lenguaje de programación o sistema.

Clase Instruccion: Esta es una clase abstracta que representa una instrucción genérica en un lenguaje de programación. Las clases que representan instrucciones específicas (como Asignacion) extienden esta clase y proporcionan implementaciones para los métodos abstractos.

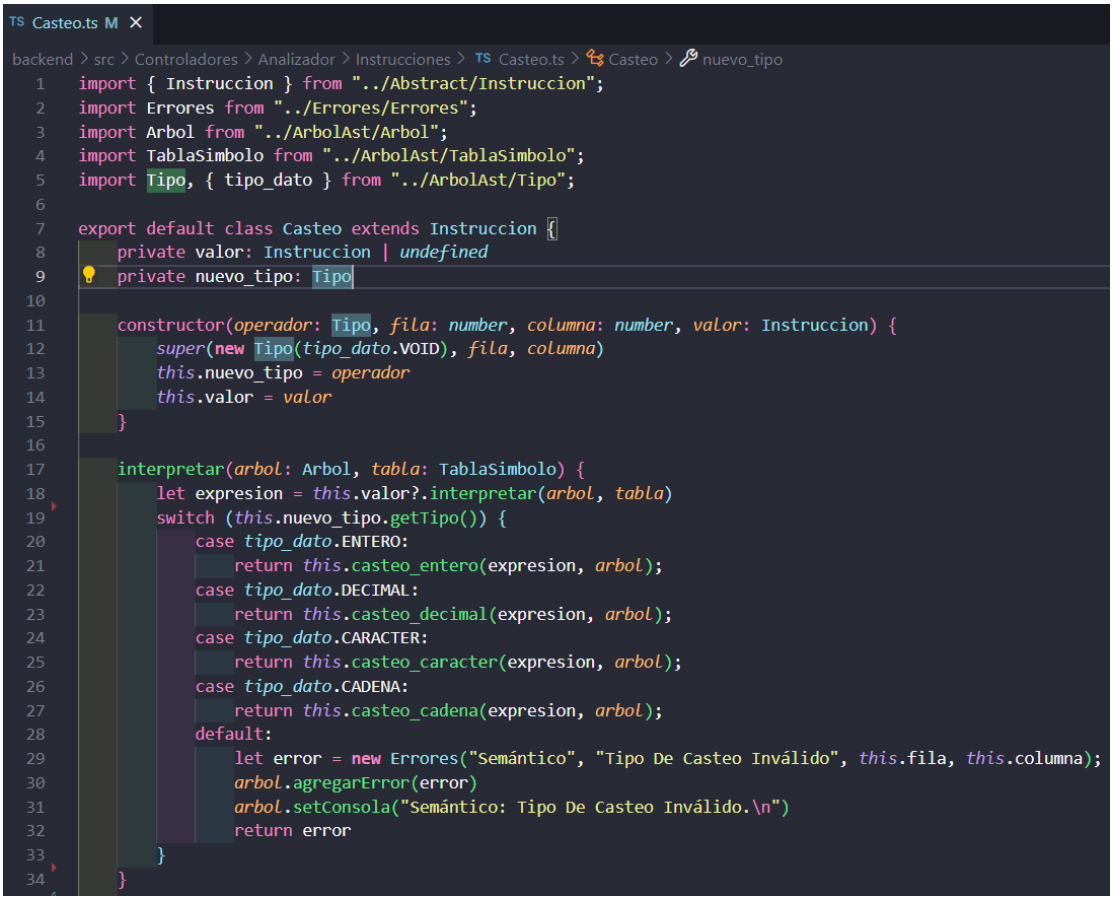
Clase Errores: Esta clase representa un error en un lenguaje de programación. Se utiliza para almacenar información sobre errores que se producen durante la interpretación de las instrucciones.

Clase Arbol: Esta clase representa un árbol de análisis, que es una estructura de datos utilizada para representar el código fuente de un programa. Contiene una lista de instrucciones y proporciona métodos para agregar contenido a la consola y manejar errores.

Clase TablaSimbolo: Esta clase representa una tabla de símbolos, que es una estructura de datos utilizada para almacenar información sobre las variables y funciones definidas en un programa.

Clase Tipo: Esta clase representa un tipo de dato en un lenguaje de programación. Se utiliza para verificar la corrección de tipo durante la interpretación de las instrucciones.

Cada una de estas clases encapsula una parte específica de la funcionalidad del sistema, y juntas permiten la interpretación de un programa escrito en el lenguaje de programación representado. Esta es la esencia de la abstracción en la programación orientada a objetos: dividir un sistema complejo en partes más pequeñas y manejables que se pueden desarrollar y entender de manera independiente.



Clase Controller

El código de la clase controller es un controlador en una aplicación Express.js. Un controlador es responsable de manejar las solicitudes HTTP entrantes y enviar respuestas HTTP. Este controlador en particular tiene tres métodos principales: interpretar_entrada, generar_reporte_tablas y generar_reporte_errores.

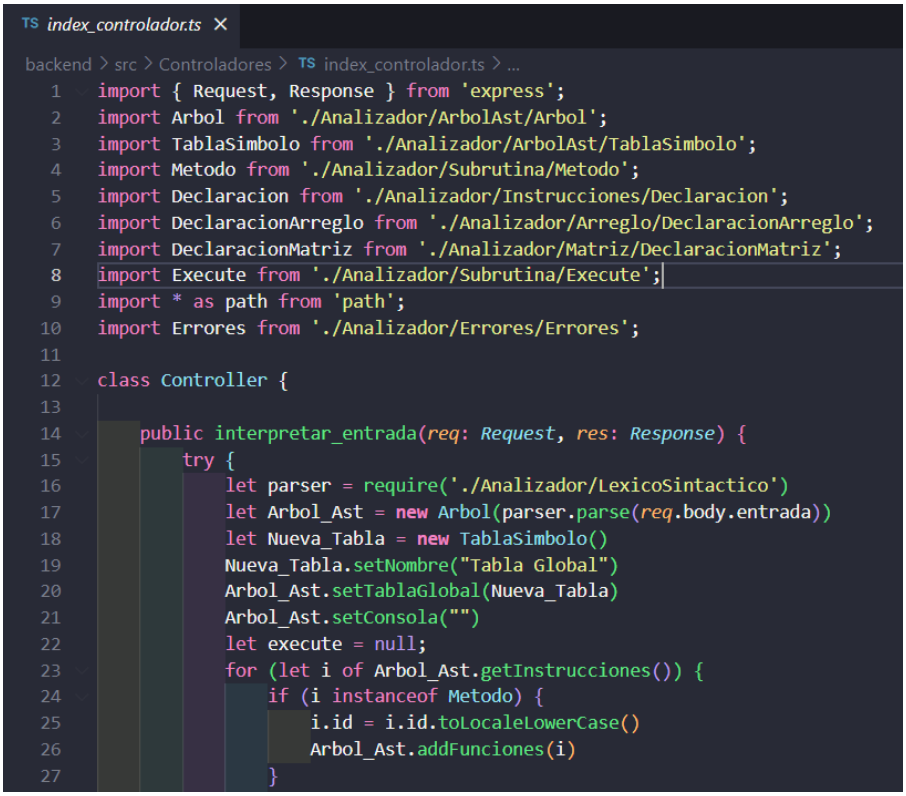
Método interpretar_entrada: Este método se encarga de interpretar el código de entrada proporcionado en la solicitud HTTP. Primero, parsea el código de entrada en un árbol de análisis abstracto (AST). Luego, crea una nueva tabla de símbolos global y la establece como la tabla global del AST. Después, recorre las instrucciones en el AST. Si la instrucción es un método, lo agrega a la lista de funciones del AST. Si la instrucción es una declaración, la interpreta. Finalmente, si hay una instrucción Execute, la interpreta. Si la interpretación de cualquier instrucción resulta en un error, lo agrega a la lista de errores del AST. Al final, envía la consola del AST como respuesta HTTP.

Método generar_reporte_tablas: Este método se encarga de generar un reporte de tablas de símbolos. El proceso para parsear el código de entrada y crear el AST es similar al método interpretar_entrada. Sin embargo, después de interpretar todas las instrucciones, genera un reporte de tablas de símbolos y envía un archivo HTML con el reporte como respuesta HTTP.

Método generar_reporte_errores: Este método se encarga de generar un reporte de errores. El proceso para parsear el código de entrada y crear el AST es similar a los otros dos métodos. Sin embargo, después de interpretar todas las instrucciones, genera un reporte de errores y envía un archivo HTML con el reporte como respuesta HTTP.

Finalmente, se exporta una instancia de la clase Controller como indexController. Esta instancia se puede utilizar en otras partes de la aplicación para manejar las rutas HTTP correspondientes.

Es importante mencionar que este controlador depende de varias otras clases (Arbol, TablaSimbolo, Metodo, Declaracion, DeclaracionArreglo, DeclaracionMatriz, Execute, Errores) para funcionar correctamente. Estas clases representan diferentes partes de un lenguaje de programación o sistema y se utilizan para interpretar el código de entrada y manejar los errores que puedan surgir durante la interpretación. Cada una de estas clases encapsula una funcionalidad específica, lo que permite una alta cohesión y un bajo acoplamiento, principios fundamentales de la programación orientada a objetos.



Clase Router

El código define una clase router que configura las rutas HTTP para una aplicación Express.js. Las rutas se definen para manejar las solicitudes POST a '/interpretar_entrada', '/generar_reporte_errores' y '/generar_reporte_tablas'.

Importación de módulos y clases: Se importan el módulo Router de Express.js y la instancia indexController de la clase Controller.

Clase router: Esta clase tiene un atributo público router que es una instancia de Router. El constructor de la clase llama al método config para configurar las rutas.

Método config: Este método configura las rutas HTTP. Define tres rutas POST que manejan las solicitudes a /interpretar_entrada, /generar_reporte_errores y /generar_reporte_tablas. Cada ruta está asociada a un método del controlador indexController.

Creación y exportación de la instancia indexRouter: Se crea una nueva instancia de la clase router y se exporta su atributo router. Este atributo es un Router de Express.js configurado con las rutas definidas en la clase router.

Este código es un ejemplo de cómo se pueden organizar y modularizar las rutas en una aplicación Express.js. Cada ruta está asociada a un método específico de un controlador, lo que permite separar la lógica de manejo de las solicitudes HTTP en diferentes métodos y clases. Esto mejora la organización del código y facilita su mantenimiento y comprensión.

Clase Servidor

Este código define una clase servidor que configura y lanza una aplicación Express.js. La aplicación utiliza varios middleware para manejar las solicitudes HTTP y define una ruta principal.

Importación de módulos y clases: Se importan varios módulos y clases necesarios para la aplicación, incluyendo express, morgan, cors, body-parser y indexRouter.

Clase servidor: Esta clase tiene un atributo público aplicacion que es una instancia de una aplicación Express.js. El constructor de la clase inicializa la aplicación, llama a los métodos configuracion y rutas para configurar la aplicación y las rutas.

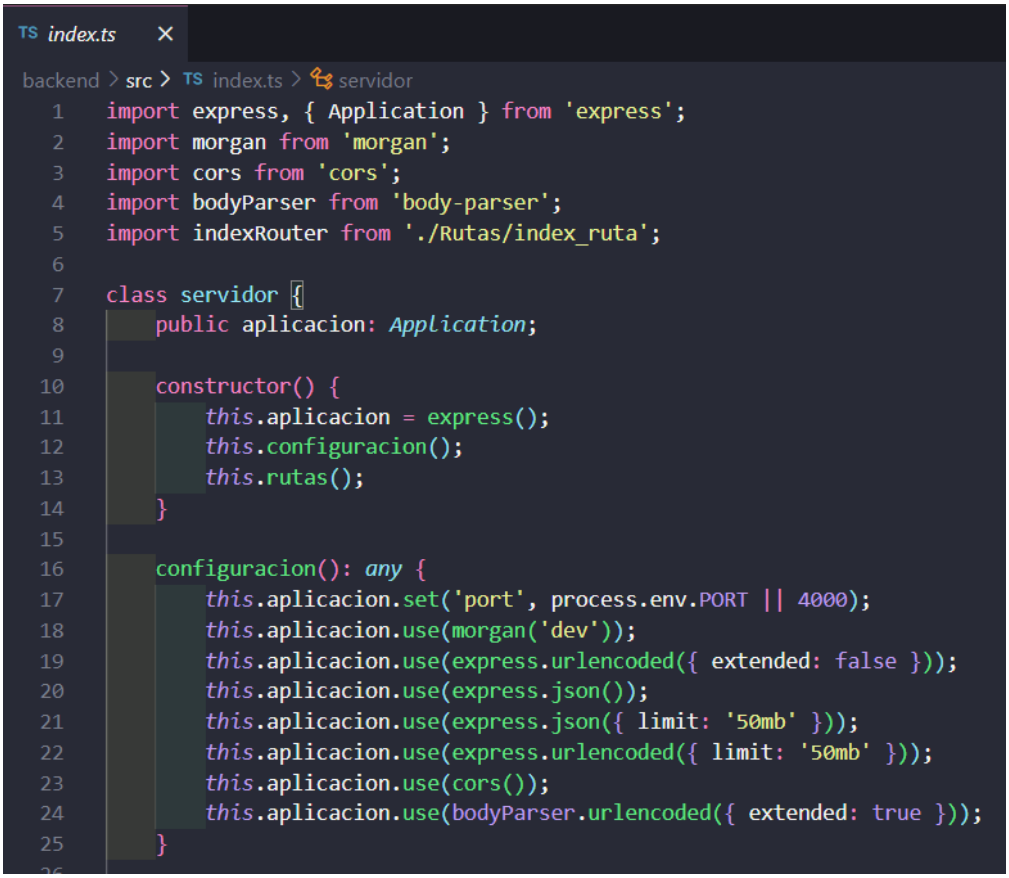
Método configuracion: Este método configura la aplicación Express.js. Establece el puerto en el que se ejecutará la aplicación, utiliza morgan para registrar las solicitudes HTTP, configura la aplicación para parsear el cuerpo de las solicitudes HTTP como JSON y como datos de formulario URL-encoded, establece un límite para el tamaño del cuerpo de las solicitudes HTTP, utiliza cors para permitir solicitudes de origen cruzado y utiliza body-parser para parsear el cuerpo de las solicitudes HTTP.

Método rutas: Este método configura las rutas de la aplicación. Utiliza indexRouter para manejar todas las solicitudes a la ruta raíz (‘/’).

Método start: Este método inicia la aplicación Express.js. Hace que la aplicación escuche en el puerto configurado y registra un mensaje en la consola cuando la aplicación comienza a escuchar en el puerto.

Creación y exportación de la instancia server: Se crea una nueva instancia de la clase servidor y se exporta. Luego, se llama al método start de la instancia para iniciar la aplicación.

Este código es un ejemplo de cómo se puede estructurar una aplicación Express.js utilizando clases y métodos para organizar la configuración y las rutas de la aplicación. Esto facilita la gestión de la aplicación y mejora la legibilidad y mantenibilidad del código.



App – Frontend

Este código define un componente de React llamado App. Este componente representa la interfaz de usuario de una aplicación web que permite a los usuarios escribir, interpretar y generar informes de código. Utiliza el editor de código Monaco para proporcionar una interfaz de usuario rica para la edición de código.

Importación de módulos y componentes: Se importan varios módulos y componentes necesarios para el componente App, incluyendo useEffect, useState, useRef de React, el componente Editor de @monaco-editor/react y el archivo CSS App.css.

Componente App: Este es un componente funcional de React. Define varias referencias (editorRef y consolaRef) para acceder a las instancias del editor de código Monaco.

Función handleEditorDidMount: Esta función se llama cuando se monta el editor de código Monaco. Guarda una referencia al editor en editorRef o consolaRef, dependiendo del valor del argumento id.

Funciones interpretar_entrada, reporte_errores y reporte_tabla: Estas funciones manejan las acciones de los botones "Ejecutar", "Errores" y "Tabla Simbolos", respectivamente. Cada función realiza una solicitud HTTP POST al servidor con el código de entrada como cuerpo de la solicitud. La respuesta del servidor se muestra en el editor de consola.

Función CargarArchivo: Esta función maneja la acción del botón de carga de archivo. Lee el contenido del archivo seleccionado y lo establece como el valor del editor de código.

Función return: Esta función devuelve el JSX (JavaScript XML) que define la interfaz de usuario del componente App. La interfaz de usuario incluye dos editores de código Monaco (uno para la entrada de código y otro para la consola), varios botones para ejecutar el código y generar informes, y un botón para cargar un archivo.

Este código es un ejemplo de cómo se puede utilizar React y el editor de código Monaco para crear una interfaz de usuario interactiva para una aplicación web de edición e interpretación de código.