MANUAL TECNICO – COMPINTERPRETER

NOMBRE: KENETH WILLARD LOPEZ OVALLE

CARNE: 202100106

DESCRIPCION

Este manual técnico proporciona una guía detallada para el desarrollo, mantenimiento y actualización del proyecto de un interpretador. Incluye información sobre la arquitectura del proyecto, el uso de las herramientas, y una descripción de clases y métodos principales.

INFORMACION GENERAL DEL PROYECTO

Este proyecto se trata de la implementación de un analizador sintáctico basado en una gramática definida para un lenguaje de programación específico. La gramática sigue el formato de **Yacc/Bison**, lo que permite la construcción de un árbol sintáctico abstracto (AST) a partir de las instrucciones proporcionadas en el código fuente. El enfoque principal del proyecto es interpretar y procesar expresiones y sentencias comunes en lenguajes de programación, como asignaciones, condicionales, ciclos, operaciones aritméticas y manipulaciones de estructuras de datos.

DESCRIPCION DEL PROYECTO

El proyecto se basa en la creación de un parser que analiza el código fuente siguiendo reglas gramaticales específicas. Estas reglas permiten reconocer diversos tipos de sentencias, como impresiones, declaraciones de variables, asignaciones, operaciones aritméticas, y estructuras de control como bucles y condicionales. El objetivo principal es generar un árbol sintáctico que pueda ser procesado por un intérprete o compilador.

Entre las características clave del parser están:

- **Instrucciones básicas**: Declaraciones de variables (incluyendo constantes), asignaciones, impresiones con print, y operaciones aritméticas y lógicas.
- **Control de flujo**: Implementación de estructuras de control como if, case, while, for, y otros tipos de bucles como do-until y loop.
- **Manejo de vectores y matrices**: Declaración y acceso a vectores unidimensionales y matrices bidimensionales.
- **Operaciones nativas**: Funciones predefinidas como lower(), upper(), round(), max(), min(), entre otras, que realizan operaciones directas sobre las expresiones.

OBJETIVOS DEL PROYECTO

1. **Implementación de un parser completo** para procesar y validar instrucciones de un lenguaje específico, construyendo un árbol sintáctico abstracto (AST) que represente el código fuente.
2. **Reconocimiento y manejo de estructuras de control**, como condicionales y ciclos, así como expresiones aritméticas y lógicas, asegurando que todas las instrucciones sigan las reglas gramaticales correctamente.
3. **Soporte para operaciones con vectores y matrices**, tanto en su declaración como en el acceso a sus elementos, permitiendo un manejo avanzado de estructuras de datos dentro del lenguaje.
4. **Implementación de funciones nativas** que puedan ser utilizadas en las expresiones del lenguaje, facilitando operaciones comunes como conversión a mayúsculas/minúsculas, redondeo de números y obtención de valores máximos y mínimos.
5. **Extensibilidad del lenguaje** para incluir más características y mejorar la interpretación o compilación, proporcionando una base sólida para futuras mejoras.

TECNOLOGIAS UTILIZADAS

1. TypeScript

Utilicé TypeScript como el lenguaje principal debido a sus capacidades avanzadas de tipado estático que permiten una mayor robustez en el desarrollo del código. TypeScript proporciona las ventajas de JavaScript, añadiendo tipificación, lo que facilita la detección temprana de errores y mejora la legibilidad del código.

2. Jison

Jison es una herramienta poderosa que permite la creación de analizadores léxicos y sintácticos. Lo empleé para generar un parser basado en gramáticas personalizadas, que permitió procesar el lenguaje específico de dominio (DSL) o las expresiones que requerían una interpretación especial. Jison facilita la conversión de una gramática BNF a un parser en JavaScript o TypeScript, lo que hace más sencillo interpretar lenguajes o generar compiladores en proyectos web.

3. React con Vite

Para el desarrollo del frontend, utilicé **React**, una biblioteca de JavaScript que permite la creación de interfaces de usuario a través de componentes reutilizables. React proporciona una estructura clara y eficiente para manejar la interacción con la interfaz de usuario, especialmente útil en aplicaciones dinámicas.

Para mejorar el rendimiento en el desarrollo del frontend, **Vite** fue la herramienta elegida. Vite es un *bundler* rápido y moderno que optimiza el tiempo de construcción y la recarga en tiempo real, lo que permite una experiencia de desarrollo más ágil comparado con otros entornos de construcción como Webpack.

4. Tailwind CSS

Implementé **Tailwind CSS** como framework de estilos para el diseño del frontend. Tailwind CSS proporciona una metodología de diseño basada en clases utilitarias, lo que permite una personalización rápida y flexible de la interfaz. Esta metodología no

solo acelera el proceso de desarrollo de estilos, sino que también facilita la creación de un diseño responsivo y coherente a lo largo de toda la aplicación.

DESCRIPCION DE METODOS Y CLASES PRINCIPALES

AST (Abstract Syntax Tree)

La clase AST representa el Árbol de Sintaxis Abstracta de la aplicación, que es fundamental para la interpretación del código y la ejecución de instrucciones. La clase se encarga de gestionar el entorno global y ejecutar las instrucciones definidas. A continuación, se detallan sus métodos y características principales:

- **Atributos:**
    - private globalEnv: Environment | null: Este atributo almacena el entorno global en el que se ejecutan las instrucciones.
    - private dotGenerator: DotGenerator: Una instancia de DotGenerator, que se utiliza para generar una representación visual del AST en formato DOT.
    - private instructions: Instruction[]: Un arreglo que contiene todas las instrucciones a ser ejecutadas.
- **Constructor:**
    - constructor(private instructions: Instruction[]): Inicializa la clase con un conjunto de instrucciones y crea una instancia de DotGenerator.
- **Métodos:**
    - interpreter(): string[]:
        - Este método es responsable de interpretar y ejecutar las instrucciones del AST. Se divide en dos fases:

1. **Registro de funciones:** Se recorren las instrucciones para registrar aquellas que son funciones en el entorno global.
2. **Ejecución de instrucciones:** Se ejecutan todas las instrucciones restantes, excluyendo las funciones ya registradas. Los resultados se almacenan en la consola.

- Retorna un arreglo de cadenas con los mensajes generados durante la ejecución.

- generateDot(): string:
  - Este método genera la representación DOT del árbol de sintaxis abstracta.
  - Crea un nodo raíz llamado "INSTRUCCIONES" y recorre todas las instrucciones, generando nodos y conectándolos a través del DotGenerator.
  - Retorna el código DOT generado.
- getGlobalEnvironment(): Environment | null:
  - Devuelve el entorno global después de la ejecución del AST, lo que permite acceder a las variables y funciones definidas.

Función setConsole

- **Descripción:**
  - setConsole(value: any): Esta función permite agregar mensajes a la consola simulando la salida generada durante la ejecución del programa.
  - Utiliza el arreglo consola para almacenar los mensajes, facilitando su visualización al final de la interpretación.

Tipo Result

El tipo Result se utiliza para encapsular el resultado de una expresión evaluada en el contexto de la interpretación del código. A continuación, se describen sus características:

- **Atributos:**
    - value: any: Este atributo almacena el valor resultante de la expresión evaluada. Puede ser de cualquier tipo de dato.
    - DataType: DataType: Este atributo indica el tipo de dato del resultado, utilizando la enumeración DataType.

Enumeración DataType

La enumeración DataType define los diferentes tipos de datos que pueden ser utilizados en la interpretación. Esto permite clasificar y manejar adecuadamente los valores durante la ejecución del programa. Los tipos de datos definidos son:

- **Tipos de Datos:**
    - ENTERO (0): Representa números enteros.
    - DECIMAL (1): Representa números decimales o flotantes.
    - BOOLEANO (2): Representa valores booleanos (true o false).
    - CHAR (3): Representa un único carácter.
    - STRING (4): Representa una cadena de texto.
    - NULO (5): Representa un valor nulo o vacío.
    - RETURN (6): Indica un valor devuelto por una función.
    - BREAK (7): Indica una interrupción en la ejecución, como en un ciclo.
    - CONTINUE (8): Indica que se debe saltar a la siguiente iteración en un ciclo.
    - VOID (9): Indica que no hay valor o que no se devuelve nada.
    - ARRAY (10): Representa un arreglo o colección de valores.

ENVIRONMENT

La clase Environment es esencial para la gestión del contexto de ejecución en la aplicación. Se encarga de almacenar y recuperar variables y funciones, así como de manejar entornos anidados. A continuación, se detallan sus métodos y características principales:

- **Atributos:**
    - private variables: Map<string, Symbol>: Este atributo almacena las variables del entorno, donde la clave es el nombre de la variable y el valor es un objeto Symbol que representa la información de la variable.
    - private funciones: Map<string, Function>: Un mapa que contiene las funciones definidas en el entorno, permitiendo su acceso y ejecución.
    - private parent: Environment | null: Este atributo hace referencia al entorno padre, lo que permite la creación de entornos anidados y la búsqueda de variables y funciones en niveles superiores.
- **Constructor:**
    - constructor(parent: Environment | null): Inicializa la clase, permitiendo la opción de asociar un entorno padre, lo que facilita la creación de entornos anidados.
- **Métodos:**
    - setVariable(name: string, symbol: Symbol): void:
        - Este método permite definir o actualizar una variable en el entorno.
        - Si la variable ya existe, se actualiza su valor; si no, se añade como una nueva variable.
    - getVariable(name: string): Symbol | null:
        - Este método busca una variable en el entorno actual y, si no se encuentra, la busca en el entorno padre.
        - Retorna el símbolo correspondiente a la variable o null si no existe.
    - setFuncion(name: string, funcion: Function): void:
        - Permite registrar una función en el entorno actual, almacenando la relación entre el nombre de la función y su implementación.

- getFuncion(name: string): Function | null:
    - Busca una función en el entorno actual y, de no encontrarla, la busca en el entorno padre.
    - Retorna la función correspondiente o null si no existe.
- getAllSymbols(): Symbol[]:
    - Este método recopila todos los símbolos (variables y funciones) en el entorno actual y en sus entornos padres.
    - Retorna un arreglo con todos los símbolos encontrados, útil para depuración y análisis.
- logError(message: string): void:
    - Registra errores semánticos en el entorno, proporcionando información sobre problemas encontrados durante la ejecución.

Symbol

La clase Symbol representa un símbolo en el entorno de ejecución de la aplicación. Este símbolo puede ser una variable, función, clase, entre otros. Almacena información fundamental sobre el símbolo, incluyendo su identificador, valor, tipo de dato y su posición en el código. A continuación, se detallan sus métodos y características principales:

- **Atributos:**
    - private id: string: Identificador (nombre) del símbolo.
    - private valor: any: Valor almacenado en el símbolo (puede ser variable, función, etc.).
    - public DataType: DataType: Tipo de dato asociado al símbolo (ej. ENTERO, BOOLEANO, STRING, etc.).

- private linea: number: Línea de código donde se declara el símbolo (para manejo de errores y depuración).
- private columna: number: Columna de código donde se declara el símbolo (para manejo de errores y depuración).
- private isConst: boolean: Indica si el símbolo es una constante.

- **Constructor:**
  - constructor(id: string, valor: any, DataType: DataType, linea: number, columna: number, isConst: boolean = false): Inicializa la clase con un identificador, valor, tipo de dato, línea y columna de declaración. Se establece isConst como false por defecto.
- **Métodos:**
  - public static crearVariableMutable(id: string, valor: any, DataType: DataType, linea: number, columna: number): Symbol:
    - Crea una nueva instancia de Symbol que representa una variable mutable.
    - Retorna el símbolo creado.
  - public static crearVariableConstante(id: string, valor: any, DataType: DataType, linea: number, columna: number): Symbol:
    - Crea una nueva instancia de Symbol que representa una constante.
    - Retorna el símbolo creado.
- **Getters:**
  - public getValor(): any: Retorna el valor almacenado en el símbolo.
  - public getId(): string: Retorna el identificador (nombre) del símbolo.
  - public getTipo(): string: Retorna el tipo de dato asociado al símbolo.
  - public getLinea(): number: Retorna la línea de código donde se define el símbolo.
  - public getColumna(): number: Retorna la columna de código donde se define el símbolo.

- **Setters:**
    - public setValor(v: Result): void: Asigna un nuevo valor al símbolo, verificando que el tipo de dato coincida con el tipo de dato del símbolo. Si el símbolo es constante, se lanza un error.
    - public setId(nuevoId: string): void: Asigna un nuevo identificador (ID) al símbolo.
    - public setTipo(nuevoTipo: DataType): void: Asigna un nuevo tipo de dato al símbolo.
    - public setLinea(nuevaLinea: number): void: Asigna una nueva línea de declaración al símbolo.
    - public setColumna(nuevaColumna: number): void: Asigna una nueva columna de declaración al símbolo.
- **Otros Métodos:**
    - public esConstante(): boolean: Retorna un booleano que indica si el símbolo es una constante.

Clase Arreglo<T>

La clase Arreglo<T> permite gestionar estructuras de datos genéricas en forma de arreglos unidimensionales o bidimensionales. A continuación, se describen sus principales atributos y métodos:

Atributos:

- **tipo: DataType**: Define el tipo de datos que el arreglo almacenará, por ejemplo, enteros, cadenas o booleanos. Este tipo es esencial para garantizar la coherencia en los valores almacenados.
- **id: string**: Representa el identificador único del arreglo. Se almacena en minúsculas para evitar diferencias entre identificadores con mayúsculas y minúsculas.
- **valores: T[][] | T[]**: Almacena los valores del arreglo. Este atributo puede representar tanto un arreglo unidimensional como bidimensional, según las necesidades.
- **dimensiones: number[]**: Indica las dimensiones del arreglo. Puede ser un arreglo unidimensional o bidimensional.

Constructor:

- **constructor**: Inicializa un arreglo con su identificador, tipo de datos, dimensiones y valores iniciales. El identificador siempre se guarda en minúsculas.

Métodos:

- **getTipo()**: Devuelve el tipo de datos del arreglo, permitiendo conocer el tipo de los elementos almacenados.
- **getLength(dimension)**: Retorna la longitud del arreglo en una dimensión específica. Para arreglos unidimensionales, devuelve su longitud total; para bidimensionales, puede devolver la longitud de las filas o columnas.
- **setTipo(tipo)**: Permite cambiar el tipo de datos del arreglo.
- **getId()**: Devuelve el identificador del arreglo, que es su nombre único.
- **setId(id)**: Asigna un nuevo identificador al arreglo, convirtiéndolo a minúsculas para mantener consistencia.
- **getValores()**: Devuelve los valores almacenados en el arreglo.
- **setValores(valores)**: Asigna nuevos valores al arreglo, permitiendo modificar el contenido de las celdas.
- **getDimensiones()**: Retorna las dimensiones del arreglo (1D o 2D).

- **setDimensiones(dimensiones)**: Define las nuevas dimensiones del arreglo, permitiendo modificar su estructura.
- **getValor(index1, index2)**: Obtiene el valor en una posición específica del arreglo. Si es unidimensional, solo requiere un índice; si es bidimensional, requiere dos índices (fila y columna).
- **setValor(index1, valor, index2)**: Asigna un nuevo valor a una posición específica en el arreglo. Verifica que el tipo de dato sea compatible con el definido para el arreglo.
- **inicializarUnidimensional(tamaño, valorPorDefecto)**: Método estático que inicializa un arreglo unidimensional con un tamaño determinado y un valor por defecto.
- **inicializarBidimensional(filas, columnas, valorPorDefecto)**: Método estático que inicializa un arreglo bidimensional, estableciendo el número de filas, columnas y un valor por defecto para cada celda.

Otros métodos:

- **tipoJavascriptADataType(valor)**: Función auxiliar que convierte los tipos de datos de JavaScript a los valores del enum DataType. Esta función es fundamental para validar que los valores ingresados coincidan con el tipo de dato definido para el arreglo.

Clase Instruction

La clase Instruction es una clase abstracta que define la estructura básica para cualquier instrucción dentro de un entorno de ejecución. Todas las instrucciones específicas que se implementen en el lenguaje o aplicación deben heredar de esta clase y proporcionar implementaciones concretas para los métodos abstractos. A continuación, se detallan sus atributos y métodos principales:

Atributos:

- **linea: number**: Almacena el número de línea en el código fuente donde se encuentra la instrucción. Este atributo es útil para depuración y manejo de errores.
- **columna: number**: Almacena el número de columna en el código fuente donde se encuentra la instrucción. Similar al atributo linea, es utilizado para el rastreo de errores y depuración.

Constructor:

- **constructor(linea: number, columna: number)**:

Inicializa la clase Instruction con los valores de línea y columna del código fuente donde se encuentra la instrucción. Estos valores son esenciales para identificar la posición de la instrucción en el código fuente.

Métodos Abstractos:

- **execute(entorno: Environment): any**

Este es un método abstracto que debe ser implementado por cualquier clase que herede de Instruction. Su función es ejecutar la lógica específica de la instrucción en un entorno particular.

    - **entorno**: Representa el contexto o ambiente en el que la instrucción se ejecuta. El Environment contiene variables, funciones y otros elementos necesarios para la ejecución.
    - **Retorna**: El resultado de la ejecución de la instrucción. Dependiendo de la instrucción, el retorno puede ser un valor, una estructura de datos o algún efecto en el entorno.
- **generateNode(dotGenerator: DotGenerator): string**

Método abstracto que genera un nodo gráfico para representar la instrucción dentro de un árbol de sintaxis abstracta (AST) en formato DOT. Cada instrucción debe implementar este método para generar su representación visual.

- **dotGenerator**: Es una instancia de DotGenerator, que se utiliza para crear nodos y conexiones dentro del AST.
- **Retorna**: Un identificador en forma de cadena que representa el nodo generado en el formato DOT. Este nodo es parte del AST y permite visualizar la estructura de la instrucción dentro del código.

Clase Expression

La clase Expression es una clase abstracta que representa una unidad de código que, al evaluarse, devuelve un valor. Las expresiones son componentes fundamentales en el lenguaje de programación y pueden incluir operaciones matemáticas, lógicas, o cualquier otra evaluación que retorne un resultado. Esta clase proporciona una estructura base que las subclases deben implementar. A continuación, se detallan sus atributos y métodos principales:

Atributos:

- **linea: number**: Almacena el número de línea en el código fuente donde se encuentra la expresión. Este atributo es útil para el manejo de errores y la depuración de código.
- **columna: number**: Almacena el número de columna en el código fuente donde se encuentra la expresión. Junto con el atributo linea, permite identificar con precisión la ubicación de la expresión en el código.

Constructor:

- **constructor (linea: number, columna: number)**:

Inicializa la expresión con los valores de línea y columna, que representan su ubicación en el código fuente. Esto es crucial para proporcionar información detallada en caso de errores y para fines de depuración.

Métodos Abstractos:

- **execute(entorno: Environment): Result**

Este método abstracto debe ser implementado por todas las subclases que hereden de Expression. Su objetivo es evaluar la expresión en el entorno dado y devolver un resultado que incluye el valor y el tipo de dato evaluado.

    - **entorno**: El entorno actual donde se evaluará la expresión. Este entorno contiene variables, funciones y otros elementos necesarios para la evaluación.
    - **Retorna**: Un objeto de tipo Result que contiene tanto el valor resultante como el tipo de dato de la evaluación.
- **generateNode(dotGenerator: DotGenerator): string**

Método abstracto que genera un nodo gráfico en formato DOT, utilizado para la representación visual de la expresión en un árbol de sintaxis abstracta (AST). Cada expresión debe implementar este método para describir cómo se representará en un AST.

    - **dotGenerator**: Es una instancia de DotGenerator que se utiliza para crear nodos y conexiones en el AST.
    - **Retorna**: Un identificador en forma de cadena que representa el nodo generado en el formato DOT. Este nodo forma parte del AST, que visualiza la estructura de la expresión.

Clase Declaration

La clase Declaration representa la declaración de variables, permitiendo definir una o varias variables, ya sean mutables o inmutables, con o sin una expresión de inicialización. Esta clase hereda de Instruction y se encarga de gestionar tanto la evaluación de expresiones como la asignación de valores iniciales a las variables. A continuación, se detallan sus atributos y métodos principales:

Atributos:

- **DataType: DataType**: Define el tipo de dato de las variables que se están declarando (por ejemplo, ENTERO, DECIMAL, BOOLEANO, etc.). Esto asegura que las variables sean compatibles con el tipo de datos especificado.
- **ids: string[]**: Una lista de identificadores de las variables que serán declaradas. Este atributo permite declarar múltiples variables en una sola instrucción.
- **exp: Expression | null**: Representa la expresión opcional que se evaluará y asignará como valor inicial a las variables. Si no se proporciona una expresión, se asignará un valor por defecto según el tipo de dato.

Constructor:

- **constructor(DataType, ids, exp, line, column)**:

Inicializa una declaración con el tipo de dato, los identificadores de las variables, una expresión opcional, y la línea y columna donde se encuentra en el código fuente.

Métodos:

- **execute(environment: Environment)**

Este método ejecuta la lógica de la declaración de variables dentro de un entorno específico. Si se proporciona una expresión, primero se evalúa y luego se asigna su resultado a cada una de las variables declaradas. Si no se proporciona expresión, se asigna un valor por defecto basado en el tipo de dato de la variable.

    - **environment**: El entorno en el que se guardan las variables declaradas.
- **evaluateExpression(environment: Environment): Result**

Evalúa la expresión de inicialización de la variable (si existe). Si no hay expresión, asigna un valor por defecto basado en el tipo de dato. También verifica si el tipo de dato de la expresión coincide con el tipo de la variable; de lo contrario, lanza un error semántico.

    - **Retorna**: Un objeto Result que contiene el valor y el tipo de dato resultante de la evaluación de la expresión.

- **getDefaultValueForType(): Result**

Devuelve el valor por defecto según el tipo de dato de la variable. Esto es útil cuando no se proporciona una expresión de inicialización.

    - **Retorna**: Un objeto Result con el valor por defecto correspondiente al tipo de dato (ENTERO, DECIMAL, BOOLEANO, etc.).
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en el formato DOT para representar gráficamente la declaración en un árbol de sintaxis abstracta (AST). Crea un nodo para la declaración, los identificadores y, si hay una expresión, un nodo para el valor de la expresión.

    - **dotGenerator**: Instancia de DotGenerator que ayuda a generar nodos y conexiones para el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo de la declaración en el AST.

Clase ConstantDeclaration

La clase ConstantDeclaration representa la declaración de variables constantes en un entorno. A diferencia de las variables mutables, las constantes no pueden cambiar su valor una vez asignado. Esta clase permite declarar una o varias constantes con o sin una expresión inicial que se evaluará para asignar el valor. Hereda de la clase Instruction y es parte del conjunto de instrucciones dentro de un lenguaje interpretado.

Atributos:

- **DataType: DataType**: Define el tipo de dato de las constantes a declarar. Este tipo puede ser de diferentes categorías como ENTERO, DECIMAL, BOOLEANO, CHAR, STRING, entre otros.

- **ids: string[]**: Una lista de identificadores para las constantes que se van a declarar. Esta lista permite declarar múltiples constantes en una sola instrucción.
- **exp: Expression | null**: Representa la expresión opcional que será evaluada para asignar su valor a la constante. Si no se proporciona ninguna expresión, se asignará un valor por defecto basado en el tipo de dato.

Constructor:

- **constructor(DataType, ids, exp, line, column)**:

Inicializa una declaración de constantes con el tipo de dato, los identificadores de las variables, la expresión opcional, y la línea y columna en el código fuente donde ocurre la declaración.

Métodos:

- **execute(environment: Environment)**

Este método ejecuta la lógica de declaración de constantes en un entorno específico. Verifica si la constante ya ha sido declarada previamente en el mismo entorno. Si no existe, guarda la constante con el valor asignado por la expresión evaluada o con un valor por defecto si no hay expresión.

    - **environment**: El entorno en el que se almacenan las constantes declaradas.
- **evaluateExpression(environment: Environment): Result**

Evalúa la expresión de inicialización de la constante (si existe). Si no hay una expresión, asigna un valor por defecto basado en el tipo de dato especificado. Además, verifica que el tipo de dato de la expresión evaluada coincida con el tipo de la constante; en caso de discrepancia, se genera un error semántico.

    - **Retorna**: Un objeto Result que contiene el valor y el tipo de dato resultante de la evaluación de la expresión o el valor por defecto.
- **getDefaultValueForType(): Result**

Devuelve un valor por defecto según el tipo de dato de la constante. Esta función es utilizada en caso de que no se proporcione una expresión inicial.

    - **Retorna**: Un objeto Result que contiene el valor por defecto para el tipo de dato (ENTERO, DECIMAL, BOOLEANO, etc.).
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la declaración de la constante en un árbol de sintaxis abstracta (AST). Se crean nodos para la declaración, los identificadores y, si hay una expresión, para el valor evaluado.

    - **dotGenerator**: Es una instancia de DotGenerator que se utiliza para crear nodos y conexiones dentro del AST.
    - **Retorna**: Un identificador en formato string que representa el nodo de la declaración de la constante en el AST.

Clase Errors

La clase Errors gestiona los errores que pueden ocurrir durante el análisis léxico, sintáctico o semántico de un programa. Estos errores son fundamentales para la depuración y permiten identificar problemas en el código de manera eficiente. A continuación, se detallan los atributos y métodos principales de la clase:

Atributos:

- **errors: Errors[]**: Es una lista estática que almacena todos los errores generados durante la ejecución. Al ser estática, permite el acceso global a los errores en cualquier parte del programa.

- **type: string**: Representa el tipo de error. Puede ser léxico, sintáctico, semántico, entre otros. Este atributo es importante para categorizar los errores según su naturaleza.
- **description: string**: Proporciona una descripción detallada del error, lo que permite al desarrollador comprender la causa y la ubicación del problema.
- **file: number**: Indica la línea en el archivo donde se detectó el error, lo cual facilita la localización del problema en el código fuente.
- **column: number**: Especifica la columna exacta dentro de la línea donde ocurrió el error, proporcionando mayor precisión en la ubicación del problema.

Constructor:

- **constructor(type: string, description: string, file: number, column: number)**: Inicializa un nuevo error con su tipo, descripción, y la ubicación en el código (línea y columna).

Métodos Getters:

- **getType()**:

Devuelve el tipo de error, permitiendo conocer si es un error léxico, sintáctico o semántico.

- **getDescription()**:

Devuelve la descripción del error, proporcionando una explicación detallada del problema.

- **getFile()**:

Retorna la línea del archivo donde ocurrió el error.

- **getColumn()**:

Retorna la columna exacta donde ocurrió el error en la línea.

Métodos Setters:

- **setType(type: string)**:

Permite modificar el tipo de error.

- **setDescription(description: string)**:

Permite cambiar la descripción del error.

- **setFile(file: number)**:

Modifica la línea donde ocurrió el error.

- **setColumn(column: number)**:

Actualiza la columna donde se detectó el error.

Métodos Estáticos:

- **addError(type: string, description: string, file: number, column: number)**

Agrega un nuevo error a la lista estática de errores. Este método permite centralizar la gestión de errores, almacenándolos de manera global.

- **getErrors()**

Devuelve la lista completa de errores almacenados. Este método es útil para acceder a todos los errores generados hasta el momento.

- **clearErrors()**

Limpia la lista de errores. Este método es ideal cuando se necesita reiniciar la gestión de errores, por ejemplo, al comenzar una nueva fase de análisis.

- **printErrors()**

Retorna una representación en formato de texto de todos los errores almacenados. Este formato es útil para mostrar los errores en consola o en archivos de logs.

- **getErrorsAsJSON()**
- Devuelve la lista de errores en formato JSON. Esto facilita la integración con sistemas que requieren manejar los errores en formato estructurado, como herramientas de depuración o interfaces gráficas.

Clase Access

La clase Access representa el acceso a una variable en el entorno de ejecución. Esta clase hereda de Expression, lo que indica que es evaluable y devuelve un valor. Su principal función es buscar una variable por su nombre en el entorno actual y devolver su valor y tipo de dato. Si la variable no se encuentra, se genera un error semántico.

Atributos:

- **id: string**: Representa el nombre de la variable a la que se desea acceder en el entorno de ejecución. Este atributo se utiliza para buscar la variable en el entorno y obtener su valor.

Constructor:

- **constructor (id: string, line: number, column: number)**:

Inicializa la instancia de Access con el identificador de la variable (id), y la posición en el código fuente (línea y columna) para ayudar en la depuración y manejo de errores.

Métodos:

- **execute(environment: Environment): Result**

Este método busca la variable en el entorno de ejecución. Si la variable existe, devuelve su valor y tipo de dato. En caso de que la variable no exista en el entorno actual ni en los entornos padres, se genera un error semántico.

    - **environment**: El entorno de ejecución donde se buscará la variable. Puede ser el entorno actual o alguno de los entornos superiores si no se encuentra en el actual.

    - **Retorna**: Un objeto Result que contiene el valor de la variable y su tipo de dato (value y DataType).
    - **Lanza**: Un error semántico si la variable no existe en el entorno.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT que representa el acceso a la variable en un árbol de sintaxis abstracta (AST). Este nodo muestra el identificador de la variable accedida.

    - **dotGenerator**: Es una instancia de DotGenerator que se utiliza para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para el acceso a la variable.

Clase Arithmetic

La clase Arithmetic representa una expresión aritmética en el lenguaje, permitiendo realizar operaciones entre dos expresiones (o una, en el caso de la negación). Esta clase hereda de Expression, lo que indica que puede evaluarse y devolver un resultado. Maneja varias operaciones aritméticas como suma, resta, multiplicación, división, potencia, raíz y módulo. A continuación, se describen sus atributos y métodos principales:

Atributos:

- **left: Expression**: Representa el operando izquierdo de la operación aritmética. Es una expresión evaluable.
- **right: Expression | null**: Representa el operando derecho de la operación, si es necesario. Algunas operaciones como la negación unaria no requieren este valor.
- **operator: ArithmeticOption**: Define la operación aritmética a realizar. Los posibles valores de este operador están definidos en el enum ArithmeticOption (SUMA, RESTA, MULTIPLICACIÓN, DIVISIÓN, POTENCIA, RAÍZ, MÓDULO, y NEGACIÓN).

Constructor:

- **constructor(left: Expression, right: Expression | null, operator: ArithmeticOption, line: number, column: number)**:

Inicializa una instancia de la clase Arithmetic con el operando izquierdo, el operando derecho (si es aplicable), el tipo de operación aritmética y la posición en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(entorno: Environment): Result**

Evalúa la expresión aritmética, ejecutando las subexpresiones (operandos) y aplicando el operador aritmético correspondiente. Si la operación es inválida según los tipos de datos, se lanza un error semántico.

    - **entorno**: El entorno de ejecución donde se evaluarán las expresiones.
    - **Retorna**: Un objeto Result que contiene el valor resultante de la operación y su tipo de dato.
    - **Lanza**: Un error semántico si la operación no es válida para los tipos de datos proporcionados, como en el caso de una división por cero.
- **ensureRightValue(rightValue: Result | null, operator: string)**

Verifica que el operando derecho no sea nulo para las operaciones que lo requieran. Si el operando es nulo, se genera un error.

- **performOperation(leftValue: Result, rightValue: Result | null, dominanceMatrix: DataType[][], operator: string): Result**

Realiza la operación aritmética entre los operandos y devuelve el resultado. Utiliza una matriz de dominancia para validar los tipos de datos de los operandos. Si la operación no es válida, se lanza un error semántico.

    - **leftValue**: Valor del operando izquierdo.
    - **rightValue**: Valor del operando derecho (si existe).
    - **dominanceMatrix**: Matriz que define la validez de la operación para los tipos de datos involucrados.
    - **operator**: El operador aritmético a aplicar (por ejemplo, "+", "-", "*", etc.).

    - **Retorna**: Un objeto Result con el resultado de la operación y su tipo de dato.
    - **Lanza**: Un error semántico si los tipos de datos no permiten la operación.
- **applyOperator(left: number, right: number, operator: string): number**

Aplica el operador aritmético a los operandos sin utilizar eval, manejando operaciones como suma, resta, multiplicación, división, potencia, módulo, y raíz.

    - **left**: Valor numérico del operando izquierdo.
    - **right**: Valor numérico del operando derecho.
    - **operator**: El operador aritmético en forma de cadena.
    - **Retorna**: El resultado de la operación entre los dos operandos.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT que representa la operación aritmética en un árbol de sintaxis abstracta (AST). Crea nodos para los operandos y el operador aritmético, y los conecta.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para la operación aritmética.

Clase Basic

La clase Basic representa una expresión básica dentro del lenguaje, como un valor literal. Estos valores pueden ser números enteros, decimales, cadenas, caracteres o booleanos. Al heredar de la clase Expression, la clase Basic es evaluable y puede devolver un resultado basado en el tipo de dato que representa.

Atributos:

- **value: string**: Representa el valor de la expresión como una cadena de texto. Este valor se convertirá al tipo de dato correspondiente durante la ejecución.
- **type: DataType**: Define el tipo de dato de la expresión. Puede ser ENTERO, DECIMAL, BOOLEANO, STRING, CHAR o NULO.

Constructor:

- **constructor(value: string, type: DataType, line: number, column: number)**:

Inicializa una expresión básica con el valor literal, el tipo de dato, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(): Result**

Este método evalúa la expresión básica y convierte el valor almacenado como cadena al tipo de dato correspondiente. Dependiendo del tipo de dato (ENTERO, DECIMAL, BOOLEANO, STRING, CHAR), se aplica la conversión adecuada.

  - **Retorna**: Un objeto Result que contiene el valor convertido al tipo correspondiente y el tipo de dato.

Conversiones específicas:

  - **ENTERO**: Convierte la cadena a un número entero (parseInt).
  - **DECIMAL**: Convierte la cadena a un número decimal (parseFloat).
  - **BOOLEANO**: Convierte la cadena a un valor booleano (basado en "true" o "false").
  - **CHAR**: Devuelve el valor como un carácter.
  - **STRING**: Devuelve el valor tal como está.
  - **NULO**: Retorna un valor nulo si el tipo no es reconocido.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT para representar la expresión básica dentro de un árbol de sintaxis abstracta (AST). El nodo contiene el valor y el tipo de dato de la expresión.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la expresión básica.

Clase Logical

La clase Logical representa una operación lógica en el lenguaje, permitiendo realizar evaluaciones basadas en operadores lógicos como AND, OR y NOT. Hereda de la clase Expression, lo que significa que es evaluable y devuelve un valor booleano. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **left: Expression**: Representa el operando izquierdo de la operación lógica. Es una expresión evaluable que debe devolver un valor booleano.
- **right: Expression | null**: Representa el operando derecho de la operación lógica. Algunas operaciones como NOT no requieren este operando, por lo que puede ser null.
- **operator: LogicalOption**: Define el operador lógico a aplicar. Puede ser uno de los siguientes valores: AND, OR, o NOT, que están definidos en el enum LogicalOption.

Constructor:

- **constructor(left: Expression, right: Expression | null, operator: LogicalOption, line: number, column: number)**:

Inicializa una instancia de la clase Logical con los operandos (izquierdo y, opcionalmente, derecho), el operador lógico, y la ubicación en el código fuente (línea y columna) para fines de depuración y manejo de errores.

Métodos:

- **execute(entorno: Environment): Result**

Evalúa la operación lógica en el entorno actual. Dependiendo del operador (AND, OR, NOT), realiza la operación adecuada y retorna el resultado como un valor booleano. Si los tipos de datos de los operandos no son booleanos, lanza un error semántico.

  - **entorno**: El entorno de ejecución donde se evaluarán las expresiones lógicas.
  - **Retorna**: Un objeto Result que contiene el valor resultante de la operación lógica y su tipo de dato (BOOLEANO).
  - **Lanza**: Un error semántico si los operandos no son booleanos o si falta un operando requerido para las operaciones binarias.

Operaciones lógicas:

- **AND (&&)**: Ambos operandos deben ser verdaderos para que el resultado sea true.
- **OR (||)**: Al menos uno de los operandos debe ser verdadero para que el resultado sea true.
- **NOT (!)**: Niega el valor booleano del operando izquierdo.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación lógica en un árbol de sintaxis abstracta (AST). Crea nodos para los operandos y el operador lógico, y los conecta visualmente.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la operación lógica.

Clase MatrixAccess

La clase MatrixAccess representa el acceso a un elemento de una matriz en un entorno de ejecución. Esta clase permite evaluar las expresiones para la fila y columna de la matriz, y acceder de forma segura al valor correspondiente en la matriz. Hereda de la clase Expression, lo que indica que es evaluable. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **id: string**: Representa el nombre de la variable que contiene la matriz en el entorno de ejecución.
- **row: Expression**: Expresión que representa el índice de la fila a la que se desea acceder. Esta expresión será evaluada en tiempo de ejecución para obtener el índice correspondiente.
- **colExpr: Expression**: Expresión que representa el índice de la columna a la que se desea acceder. También será evaluada en tiempo de ejecución.

Constructor:

- **constructor(id: string, row: Expression, colExpr: Expression, line: number, column: number)**:

Inicializa una instancia de MatrixAccess con el identificador de la matriz, las expresiones de fila y columna, y la ubicación en el código fuente (línea y columna) para fines de depuración y manejo de errores.

Métodos:

- **execute(environment: Environment): Result**

Evalúa el acceso a un elemento específico de la matriz en un entorno dado. Primero busca la variable que contiene la matriz en el entorno, luego evalúa las expresiones de fila y columna para obtener los índices correspondientes. Accede al valor en la matriz utilizando el método getValor de la clase Arreglo.

    - **environment**: El entorno donde se buscará la variable que contiene la matriz.
    - **Retorna**: Un objeto Result que contiene el valor del elemento en la matriz y su tipo de dato.
    - **Lanza**: Un error semántico si la variable no es una matriz o si ocurre algún problema en el acceso, como un índice fuera de rango.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT para representar el acceso a la matriz en un árbol de sintaxis abstracta (AST). Crea nodos para las expresiones de fila y columna, y los conecta al nodo del acceso a la matriz.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para el acceso a la matriz.

Clase Relational

La clase Relational representa una operación relacional en el lenguaje, permitiendo comparar dos expresiones mediante operadores como IGUALDAD, DISTINTO, MAYOR, MENOR, MAYORIGUAL y MENORIGUAL. Esta clase hereda de Expression, lo que significa que puede evaluarse y devolver un valor booleano, indicando el resultado de la comparación. A continuación, se detallan los atributos y métodos principales de la clase:

Atributos:

- **left: Expression**: Representa la expresión evaluable del lado izquierdo de la operación relacional.
- **right: Expression**: Representa la expresión evaluable del lado derecho de la operación relacional.
- **operator: RelationalOption**: Define el tipo de operación relacional a aplicar (IGUALDAD, DISTINTO, MAYOR, MENOR, MAYORIGUAL, MENORIGUAL), valores definidos en el enum RelationalOption.

Constructor:

- **constructor(left: Expression, right: Expression, operator: RelationalOption, line: number, column: number)**:

Inicializa una instancia de la clase Relational con las expresiones a la izquierda y derecha de la operación, el operador relacional, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Evalúa la operación relacional en el entorno actual. Compara las expresiones de izquierda y derecha en función del operador relacional y devuelve un valor booleano que indica si la relación es verdadera o falsa. Si los tipos de datos no son compatibles para la comparación, lanza un error.

  - **environment**: El entorno donde se evaluarán las expresiones.
  - **Retorna**: Un objeto Result que contiene el resultado booleano de la operación y su tipo de dato (BOOLEANO).
  - **Lanza**: Un error semántico si la operación relacional no es válida para los tipos de datos proporcionados.

Operaciones soportadas:

  - **IGUALDAD**: Devuelve true si ambos operandos son iguales.

    - **DISTINTO**: Devuelve true si los operandos son diferentes.
    - **MAYOR**: Devuelve true si el operando izquierdo es mayor que el derecho (solo para números).
    - **MENOR**: Devuelve true si el operando izquierdo es menor que el derecho (solo para números).
    - **MAYORIGUAL**: Devuelve true si el operando izquierdo es mayor o igual que el derecho (solo para números).
    - **MENORIGUAL**: Devuelve true si el operando izquierdo es menor o igual que el derecho (solo para números).
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación relacional en un árbol de sintaxis abstracta (AST). Crea nodos para las expresiones izquierda y derecha, y los conecta al nodo de la operación relacional.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para la operación relacional.

Clase Ternary

La clase Ternary representa una expresión ternaria en el lenguaje, que permite seleccionar entre dos resultados posibles (una expresión "verdadera" y una "falsa") en función de una condición booleana. Esta clase hereda de Expression, lo que indica que es evaluable y devuelve un resultado. A continuación, se detallan los atributos y métodos principales de la clase.

Atributos:

- **condition: Expression**: Es la condición que se evaluará. Esta condición debe ser una expresión lógica o relacional, y su valor debe ser booleano para decidir si se evalúa la expresión "verdadera" o "falsa".
- **trueExpression: Expression**: La expresión que se ejecutará si la condición es verdadera. Esta expresión puede devolver cualquier tipo de valor.
- **falseExpression: Expression**: La expresión que se ejecutará si la condición es falsa. Similar a trueExpression, puede devolver cualquier tipo de valor.

Constructor:

- **constructor(condition: Expression, trueExpression: Expression, falseExpression: Expression, line: number, column: number)**:

Inicializa la instancia de la clase Ternary con la condición, las expresiones "verdadera" y "falsa", y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Evalúa la condición y selecciona cuál de las dos expresiones (verdadera o falsa) ejecutar. Si la condición es true, se ejecuta y retorna el resultado de trueExpression; si es false, se ejecuta y retorna el resultado de falseExpression.

  - **environment**: El entorno donde se evaluarán las expresiones.
  - **Retorna**: Un objeto Result que contiene el valor devuelto por la expresión seleccionada y su tipo de dato.
  - **Lanza**: Un error semántico si la condición no es una expresión lógica o relacional.

Proceso:

  - Evalúa la condición.
  - Si la condición es verdadera, ejecuta la expresión "verdadera".

    - Si la condición es falsa, ejecuta la expresión "falsa".
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación ternaria en un árbol de sintaxis abstracta (AST). Crea nodos para la condición, la expresión "verdadera" y la expresión "falsa", y los conecta al nodo principal de la operación ternaria.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para la expresión ternaria.

Clase VectorAccess

La clase VectorAccess representa el acceso a un elemento específico en un vector unidimensional dentro de un entorno de ejecución. Hereda de la clase Expression, lo que indica que es evaluable y devuelve un valor basado en el índice proporcionado. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **id: string**: Representa el identificador del vector en el entorno. Este es el nombre de la variable que contiene el vector.
- **index: Expression**: Expresión que será evaluada para obtener el índice del elemento dentro del vector. Este índice debe ser un valor numérico que se evaluará en tiempo de ejecución.

Constructor:

- **constructor(id: string, index: Expression, line: number, column: number)**:

Inicializa una instancia de la clase VectorAccess con el identificador del vector, la expresión que representa el índice, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Evalúa el acceso a un elemento del vector en un entorno específico. Busca la variable correspondiente al vector en el entorno, evalúa el índice, y luego intenta acceder al valor en la posición indicada. Si ocurre algún problema (por ejemplo, el índice no es un número o está fuera de rango), se lanza un error semántico.

    - **environment**: El entorno donde se buscará la variable que contiene el vector.
    - **Retorna**: Un objeto Result que contiene el valor accedido y su tipo de dato.
    - **Lanza**: Un error semántico si el identificador no es un vector, si el índice no es válido o si ocurre algún problema al acceder al vector.
- **detectDataType(value: any): DataType**

Detecta y retorna el tipo de dato del valor accedido en el vector. Esto permite manejar diferentes tipos de datos que puedan estar almacenados en el vector.

    - **value**: El valor accedido en el vector.

    - **Retorna**: El tipo de dato correspondiente (ENTERO, STRING, BOOLEANO, NULO).
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa el acceso al vector en un árbol de sintaxis abstracta (AST). Crea nodos para el identificador del vector y el índice, y los conecta para representar visualmente la operación de acceso.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para el acceso al vector.

Clase Argument

La clase Argument representa un argumento pasado a una función o método en un entorno de ejecución. Su propósito es verificar y evaluar la expresión asociada al argumento, asegurándose de que los tipos de datos sean correctos y asignando el valor evaluado a la variable correspondiente. Esta clase hereda de Instruction, lo que indica que puede ser ejecutada dentro de un contexto de ejecución. A continuación, se detallan sus atributos y métodos principales.

Atributos:

- **id: string**: Representa el nombre del argumento o parámetro, que es la variable a la que se le asignará el valor evaluado.
- **expression: Expression**: Es la expresión que se evaluará para obtener el valor que será asignado al argumento.
- **callEnv: Environment | null**: Representa el entorno en el que se ha declarado el argumento. Puede ser null si aún no se ha asociado un entorno.

Constructor:

- **constructor(id: string, expression: Expression, line: number, column: number)**:

Inicializa una instancia de Argument con el identificador del argumento, la expresión a evaluar, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(entorno: Environment)**

Este método ejecuta la asignación del valor evaluado a la variable del argumento. Primero, busca la variable en el entorno de la llamada (callEnv), luego evalúa la expresión asociada, y finalmente, verifica si el tipo de dato de la expresión coincide con el tipo de la variable. Si el tipo es correcto, se asigna el valor; si no, se lanza un error.

    - **entorno**: El entorno de ejecución donde se evalúan las expresiones.
    - **Lanza**: Un error semántico si la variable no se encuentra en el entorno o si el tipo de dato de la expresión no coincide con el de la variable.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa el argumento en un árbol de sintaxis abstracta (AST). Crea un nodo para el argumento y lo conecta con el nodo que representa la expresión evaluada.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para el argumento.
- **setEnv(v: Environment)**

Asigna un entorno de ejecución específico (callEnv) al argumento, que representa el contexto en el que se evaluará la variable asociada al argumento.

    - **v**: El entorno que se asociará al argumento.

Clase Assignment

La clase Assignment representa la operación de asignación en el lenguaje, es decir, la asignación de un valor a una variable previamente declarada. Hereda de la clase Instruction, lo que significa que se puede ejecutar dentro de un entorno de ejecución. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **id: string**: Representa el nombre de la variable a la que se le asignará el resultado de la expresión evaluada.

- **exp: Expression**: Es la expresión que será evaluada y cuyo resultado será asignado a la variable.

Constructor:

- **constructor(id: string, exp: Expression, linea: number, columna: number)**: Inicializa una instancia de la clase Assignment con el identificador de la variable, la expresión a evaluar y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment)**
  Este método ejecuta la operación de asignación en el entorno proporcionado. Busca la variable en el entorno, evalúa la expresión, y asigna el valor resultante a la variable. Si la variable es constante o no existe, lanza un error semántico.
  - **environment**: El entorno donde se realizará la búsqueda de la variable y la evaluación de la expresión.
  - **Lanza**: Un error semántico si la variable no ha sido declarada, si es constante, o si el tipo de la expresión no coincide con el tipo de la variable.

Proceso:

  - Busca la variable en el entorno. Si no se encuentra, lanza un error.
  - Verifica si la variable es constante; si lo es, lanza un error porque no puede ser modificada.
  - Evalúa la expresión proporcionada.
  - Verifica si el tipo de la variable coincide con el tipo del resultado de la expresión; si no coincide, lanza un error.
  - Asigna el valor evaluado a la variable.
- **generateNode(dotGenerator: DotGenerator): string**
  Genera un nodo gráfico en formato DOT que representa la operación de asignación en un árbol de sintaxis abstracta (AST). Crea nodos para la variable y la expresión, y los conecta.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la asignación.

Clase Average

La clase Average representa una operación que calcula el promedio de los elementos de un vector en un entorno de ejecución. Hereda de la clase Expression, lo que significa que puede evaluarse y devolver un resultado basado en el cálculo del promedio de los valores dentro del vector. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **id: string**: Representa el identificador del vector (arreglo) del cual se calculará el promedio. Este es el nombre de la variable que contiene el vector.

Constructor:

- **constructor(id: string, line: number, column: number)**:

Inicializa una instancia de la clase Average con el identificador del vector y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método evalúa la operación de calcular el promedio del vector en un entorno específico. Busca el vector en el entorno, verifica su existencia y su tipo de dato, y luego realiza el cálculo del promedio dependiendo del tipo de los elementos del vector.

  - **environment**: El entorno donde se buscará el vector y se calculará su promedio.
  - **Retorna**: Un objeto Result que contiene el valor del promedio y su tipo de dato.
  - **Lanza**: Un error semántico si:
    - El identificador no corresponde a un vector.
    - El vector está vacío (para evitar la división por cero).
    - El tipo de dato del vector no es compatible con la operación de promedio (por ejemplo, STRING).

Proceso:

  - **Verificar si el vector existe**: Si no se encuentra o no es un vector, se lanza un error.
  - **Verificar si el vector está vacío**: Si el vector está vacío, se lanza un error para evitar la división por cero.
  - **Calcular el promedio**: Dependiendo del tipo de datos del vector, se realiza el cálculo adecuado:

        - **ENTERO y DECIMAL**: Suma de los valores dividida por la cantidad de elementos.
        - **CHAR**: Suma de los valores ASCII de los caracteres dividida por la cantidad de elementos.
        - **BOOLEANO**: Se cuenta true como 1 y false como 0, y se calcula el promedio.
    - **Lanzar un error para tipos incompatibles**: Si el vector es de tipo STRING, se lanza un error ya que el promedio no es aplicable.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación de cálculo del promedio en un árbol de sintaxis abstracta (AST). Crea un nodo que identifica la operación de promedio y lo asocia con el vector correspondiente.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la operación Average.

Clase Call

La clase Call representa la llamada a una función dentro del entorno de ejecución. Su objetivo es ejecutar la función, pasar los argumentos adecuados y gestionar el entorno local de dicha función. Esta clase hereda de Expression, lo que indica que puede evaluarse y devolver un resultado. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **id: string**: Representa el identificador de la función que se va a invocar.
- **args: { id: string, value: any }[]**: Lista de argumentos pasados a la función. Cada argumento tiene un identificador (id) y un valor (value). Estos argumentos se asignarán a los parámetros correspondientes de la función.

Constructor:

- **constructor(id: string, args: { id: string, value: any }[], line: number, column: number)**:

Inicializa una instancia de la clase Call con el identificador de la función, los argumentos que se pasarán a la función, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método ejecuta la función invocada. Primero, busca la función en el entorno actual. Luego, asigna los argumentos a los parámetros de la función y ejecuta el bloque de instrucciones asociado a la función en un nuevo entorno local.

  - **environment**: El entorno donde se buscará la función y se ejecutarán las instrucciones.
  - **Retorna**: El resultado de la ejecución de la función o un valor nulo si no hay un retorno explícito.
  - **Lanza**: Un error semántico si:
    - La función no está definida.
    - Se pasa un número incorrecto de argumentos.
    - Falta un valor para un parámetro obligatorio.

Proceso:

  - **Buscar la función**: Verifica que la función exista en el entorno actual o global. Si no existe, lanza un error.

- **Verificar la cantidad de parámetros**: Si se pasan más argumentos de los necesarios, lanza un error.
- **Crear un entorno local**: Se crea un nuevo entorno para la función, donde se asignan los parámetros y se ejecutan las instrucciones de la función.
- **Asignar los argumentos a los parámetros**: Asigna los valores pasados en la llamada a los parámetros correspondientes, utilizando valores por defecto si es necesario.
- **Ejecutar las instrucciones de la función**: Ejecuta el bloque de instrucciones de la función en el entorno local.
- **Retornar el resultado**: Devuelve el valor resultante de la ejecución de la función o null si no hay un retorno explícito.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la llamada a la función en un árbol de sintaxis abstracta (AST). Crea nodos para los argumentos pasados y los conecta al nodo de la llamada a la función.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la llamada a la función.

Clase Case

La clase Case representa una opción en una instrucción switch dentro de un lenguaje de programación. Cada Case tiene una expresión asociada y un conjunto de instrucciones que se ejecutan si el valor de la expresión del switch coincide con la expresión del Case. Esta clase hereda de Instruction, lo que significa que puede ser ejecutada dentro de un entorno. A continuación, se detallan sus atributos y métodos principales.

Atributos:

- **expression: Expression**: Representa la expresión del case, que será comparada con la expresión del switch.
- **instructions: Instruction[]**: Es una lista de instrucciones que se ejecutan cuando el case es seleccionado.

Constructor:

- **constructor(expression: Expression, instructions: Instruction[], line: number, column: number)**:

Inicializa una instancia de la clase Case con la expresión a evaluar, las instrucciones asociadas y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **getExpression(): Expression**

Este método devuelve la expresión asociada al case, permitiendo obtenerla para su comparación con la expresión del switch.

  - **Retorna**: La expresión asociada al case.
- **execute(environment: Environment): any**

Ejecuta las instrucciones asociadas al case dentro del entorno proporcionado. Si una de las instrucciones contiene un break, el método devolverá ese break para que el switch pueda manejarlo.

  - **environment**: El entorno donde se ejecutarán las instrucciones del case.
  - **Retorna**: El resultado de la ejecución, o un objeto Break si se encuentra una instrucción break.

Proceso:

  - Itera sobre cada instrucción en el case.

    - Ejecuta cada instrucción en el entorno.
    - Si encuentra un break, detiene la ejecución y retorna el break para que sea manejado por el switch.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa el case en un árbol de sintaxis abstracta (AST). Crea nodos para la expresión del case y las instrucciones asociadas, y los conecta al nodo principal del case.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para el case.

Proceso:

    - Genera el nodo para la expresión del case.
    - Crea el nodo principal del case.
    - Conecta el nodo de la expresión al nodo del case.
    - Genera y conecta los nodos de cada instrucción del case al nodo principal.

Clase Cast

La clase Cast representa una operación de conversión de tipo (casting) en un entorno de ejecución. Esta operación permite cambiar el tipo de un valor a otro compatible, como convertir un entero a decimal, o un carácter a entero. Hereda de la clase Expression, lo que significa que puede ser evaluada y devolver un resultado. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **expression: Expression**: Es la expresión cuyo valor se quiere convertir a otro tipo de dato.
- **targetType: DataType**: Es el tipo de dato al que se quiere convertir el valor de la expresión (por ejemplo, ENTERO, DECIMAL, CHAR).

Constructor:

- **constructor(expression: Expression, targetType: DataType, line: number, column: number)**:

Inicializa una instancia de la clase Cast con la expresión a convertir, el tipo de dato de destino, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método realiza la conversión de tipo en función del tipo de dato de la expresión original y el tipo de dato de destino. Evalúa la expresión, y dependiendo del tipo de la expresión, llama a los métodos específicos para realizar el casting.

  - **environment**: El entorno donde se evalúa la expresión y se realiza la conversión de tipo.
  - **Retorna**: Un objeto Result que contiene el valor convertido y su tipo de dato.
  - **Lanza**: Un error semántico si el tipo de dato de la expresión no puede ser convertido al tipo de destino.

Proceso:

  - **Evaluar la expresión**: Obtiene el valor y tipo de la expresión original.
  - **Validar las combinaciones permitidas**: Dependiendo del tipo de dato de la expresión, se invoca un método especializado para realizar la conversión:

        - **castFromInt()**: Realiza conversiones desde el tipo ENTERO a otros tipos como DECIMAL, STRING o CHAR.
        - **castFromDouble()**: Convierte de DECIMAL a ENTERO o STRING.
        - **castFromChar()**: Convierte de CHAR a ENTERO o DECIMAL.
    - **Lanzar un error si la conversión no es posible**: Si el tipo de la expresión no es compatible con el tipo de destino, se lanza un error.
- **castFromInt(exprValue: Result): Result**

Realiza el casting desde un valor de tipo ENTERO a otro tipo de dato.

    - **Retorna**: El valor convertido al tipo de destino (por ejemplo, DECIMAL, STRING, CHAR).
- **castFromDouble(exprValue: Result): Result**

Realiza el casting desde un valor de tipo DECIMAL a otro tipo de dato.

    - **Retorna**: El valor convertido al tipo de destino (por ejemplo, ENTERO, STRING).
- **castFromChar(exprValue: Result): Result**

Realiza el casting desde un valor de tipo CHAR a otro tipo de dato.

    - **Retorna**: El valor convertido al tipo de destino (por ejemplo, ENTERO, DECIMAL).
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación de conversión en un árbol de sintaxis abstracta (AST). Crea un nodo para la expresión original y lo conecta con el nodo del tipo de dato de destino.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para la operación de conversión de tipo.

Clase Decrement

La clase Decrement representa una instrucción que realiza la operación de decremento (restar 1) sobre una variable en un entorno de ejecución. Esta clase hereda de Instruction, lo que significa que puede ejecutarse dentro del flujo del programa y modificar el estado de las variables en el entorno. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **id: string**: Representa el identificador de la variable que se va a decrementar. Es el nombre de la variable que debe existir en el entorno.

Constructor:

- **constructor(id: string, line: number, column: number)**:

Inicializa una instancia de la clase Decrement con el identificador de la variable, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método realiza la operación de decremento en la variable indicada. Busca la variable en el entorno, verifica que sea un valor numérico, y decrementa su valor en 1. Si la variable no existe o no es numérica, lanza un error semántico.

  - **environment**: El entorno donde se buscará la variable y se realizará el decremento.
  - **Retorna**: Un objeto Result con un valor nulo (NULO) porque es una operación de modificación de estado y no tiene un valor de retorno explícito.
  - **Lanza**: Un error semántico si la variable no ha sido declarada o si el valor de la variable no es numérico.

Proceso:

- **Buscar la variable**: Verifica que la variable exista en el entorno. Si no existe, lanza un error.
- **Verificar el tipo de dato**: Asegura que el valor actual de la variable sea numérico. Si no lo es, lanza un error.
- **Decrementar el valor**: Resta 1 al valor actual de la variable.
- **Actualizar el entorno**: Almacena el nuevo valor en el entorno, actualizando la variable.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación de decremento en un árbol de sintaxis abstracta (AST). Crea un nodo para la variable que está siendo decrementada.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la operación de decremento.

Clase Default

La clase Default representa la instrucción default en una estructura de control switch. Esta instrucción se ejecuta cuando ninguna de las expresiones de los case anteriores coincide con el valor evaluado. Hereda de Instruction, lo que indica que puede ejecutarse dentro del flujo de un programa y contiene un conjunto de instrucciones que se ejecutan en caso de ser seleccionada. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **instructions: Instruction[]**: Es una lista de instrucciones que se ejecutan cuando el default es seleccionado. Estas instrucciones representan el bloque de código que se ejecuta si no coincide ningún case.

Constructor:

- **constructor(instructions: Instruction[], line: number, column: number)**:

Inicializa una instancia de la clase Default con el conjunto de instrucciones que se ejecutarán, así como la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): any**

Este método ejecuta las instrucciones asociadas al default en el entorno proporcionado. Itera sobre cada instrucción y las ejecuta secuencialmente.

  - **environment**: El entorno donde se ejecutarán las instrucciones del bloque default.
  - **Retorna**: No devuelve un valor explícito, ya que su propósito es ejecutar un conjunto de instrucciones en un bloque default.

Proceso:

    - Itera sobre la lista de instrucciones.
    - Ejecuta cada instrucción en el entorno proporcionado.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa el bloque default en un árbol de sintaxis abstracta (AST). Crea nodos para cada instrucción dentro del bloque default y los conecta al nodo principal.

    - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
    - **Retorna**: Un identificador en formato string que representa el nodo creado para la instrucción default.

Proceso:

    - Crea el nodo principal para la instrucción default.
    - Genera los nodos correspondientes a cada instrucción dentro del bloque default.
    - Conecta estos nodos al nodo default principal.

Clase DoUntil

La clase DoUntil representa una estructura de control de flujo que ejecuta un bloque de instrucciones repetidamente hasta que una condición se evalúa como verdadera. A diferencia de otros ciclos como while, el ciclo do-until siempre ejecuta el bloque de instrucciones al menos una vez, ya que evalúa la condición al final del ciclo. Esta clase hereda de Instruction, lo que le permite ser ejecutada dentro del flujo de un programa. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **condition: Expression**: Es la condición que se evaluará al final de cada iteración del ciclo. Si la condición es verdadera, el ciclo se detiene.
- **instructions: Instruction[]**: Es una lista de instrucciones que se ejecutan en cada iteración del ciclo, hasta que la condición se evalúe como verdadera.

Constructor:

- **constructor(condition: Expression, instructions: Instruction[], line: number, column: number)**:

Inicializa una instancia de la clase DoUntil con la condición que se evaluará y el conjunto de instrucciones que se ejecutarán en cada iteración. También incluye la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment)**

Este método ejecuta el ciclo do-until en el entorno proporcionado. Se crea un nuevo entorno local para cada iteración del ciclo, se ejecutan las instrucciones, y luego se evalúa la condición. Si la condición no es booleana, lanza un error semántico.

    - **environment**: El entorno donde se ejecutarán las instrucciones y se evaluará la condición.

- **Lanza**: Un error semántico si la condición no es de tipo BOOLEANO.

Proceso:

- **Crear un nuevo entorno**: Un nuevo entorno se crea para encapsular las variables y el estado local del ciclo.
- **Ejecutar las instrucciones**: El bloque de instrucciones se ejecuta al menos una vez.
- **Evaluar la condición**: Al final de cada iteración, se evalúa la condición. Si es booleana y su valor es verdadero, el ciclo se detiene; si no, continúa.
- **Manejo de Break y Continue**: Si encuentra una instrucción Break, el ciclo se detiene inmediatamente. Si se encuentra una instrucción Continue, se pasa a la siguiente iteración del ciclo.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa el ciclo do-until en un árbol de sintaxis abstracta (AST). Crea nodos para la condición y las instrucciones, y los conecta al nodo principal del ciclo.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la estructura do-until.

Proceso:

- **Generar el nodo de la condición**: Representa la condición evaluada al final de cada iteración.
- **Crear el nodo principal**: Crea el nodo que representa el ciclo Do-Until.
- **Conectar las instrucciones**: Genera nodos para cada instrucción dentro del ciclo y los conecta al nodo principal.

Clase Echo

La clase Echo representa una instrucción que imprime el resultado de una expresión en la consola. Esta instrucción evalúa una expresión, formatea su salida y la muestra en el entorno de ejecución. La clase hereda de Instruction, lo que significa que puede ejecutarse dentro del flujo del programa. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **expression: Expression**: Es la expresión que se evaluará y cuyo resultado será impreso en la consola. Puede ser de cualquier tipo de dato compatible (como STRING, ENTERO, BOOLEANO, etc.).

Constructor:

- **constructor(expression: Expression, line: number, column: number)**:

Inicializa una instancia de la clase Echo con la expresión a imprimir y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment)**

Este método evalúa la expresión, formatea su valor y lo imprime en la consola. Si el valor es null o undefined, imprime "null". Además, maneja el formato de cadenas de texto, eliminando caracteres escapados y comillas innecesarias.

  - **environment**: El entorno donde se evalúa la expresión y se realiza la impresión.
  - **Lanza**: Captura cualquier error que ocurra durante la evaluación de la expresión y lo muestra en la consola.

Proceso:

  - **Evaluar la expresión**: Obtiene el valor de la expresión.

- **Formatear el valor**: Dependiendo del tipo de dato, limpia la cadena de caracteres escapados y convierte otros tipos de datos a su representación en string.
- **Imprimir en la consola**: Si el valor es válido, lo imprime; de lo contrario, imprime "null".
- **Manejo de errores**: Si ocurre algún error en la evaluación o en la impresión, se captura y se muestra el error en la consola.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la instrucción Echo en un árbol de sintaxis abstracta (AST). Crea nodos para la expresión que se imprime y conecta estos nodos al nodo principal Echo.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la instrucción Echo.

Proceso:

- **Generar el nodo de la expresión**: Representa la expresión que se evaluará e imprimirá.
- **Crear el nodo principal**: Crea el nodo Echo que representa la instrucción de impresión.
- **Conectar los nodos**: Conecta el nodo de la expresión al nodo principal Echo.

Clase Execute

La clase Execute representa la ejecución de un método en el entorno de ejecución de un programa. Esta instrucción actúa como el punto de entrada del programa, invocando un método específico con los parámetros proporcionados. Hereda de Instruction, lo que le permite ser ejecutada dentro del flujo del programa. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **id: string**: Es el identificador del método que se va a ejecutar.
- **parametros: { id: string, value: any }[]**: Lista de parámetros opcionales que se pasan al método. Cada parámetro tiene un identificador (id) y un valor (value).

Constructor:

- **constructor(id: string, parametros: { id: string, value: any }[], line: number, column: number)**:

Inicializa una instancia de la clase Execute con el identificador del método, los parámetros opcionales, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment)**

Este método busca el método especificado por su identificador en el entorno global, y si lo encuentra, lo ejecuta en un nuevo entorno. También permite manejar la creación de un subentorno donde se ejecutarán las instrucciones del método.

    - **environment**: El entorno donde se ejecutará el método y donde se buscará su definición.
    - **Lanza**: Un error semántico si el método no está definido en el entorno global.

Proceso:

- **Buscar el método**: El método se busca en el entorno global. Si no se encuentra, se lanza un error.
- **Crear un nuevo entorno**: Si el método existe, se crea un nuevo entorno donde se ejecutarán las instrucciones del método, manteniendo el entorno actual como padre.
- **Ejecutar las instrucciones del método**: Una vez creado el entorno, se ejecutan las instrucciones del método en el nuevo entorno.

Manejo de parámetros: Aunque los parámetros están definidos en el constructor, no se procesan explícitamente en este método. En implementaciones adicionales, podrían ser utilizados para pasar argumentos al método ejecutado.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la llamada al método en un árbol de sintaxis abstracta (AST). Crea nodos para el método y sus parámetros, y los conecta al nodo principal.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la instrucción Execute.

Proceso:

- **Crear el nodo del método**: Representa la llamada al método.
- **Agregar los nodos de parámetros**: Si hay parámetros, se crean nodos para cada uno y se conectan al nodo principal del método.

Clase For

La clase For representa una estructura de control de flujo que ejecuta un bloque de instrucciones de manera repetitiva. El ciclo for sigue la estructura clásica que incluye una inicialización, una condición que se evalúa antes de cada iteración, una actualización después de cada iteración, y un conjunto de instrucciones que se ejecutan en cada iteración. Esta clase hereda de Instruction, lo que le permite ser ejecutada dentro del flujo del programa. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **initialization: Instruction**: Representa la instrucción de inicialización, que normalmente define e inicializa una variable de control del ciclo.
- **condition: Expression**: Es una expresión booleana que se evalúa antes de cada iteración del ciclo. Si la condición es falsa, el ciclo se detiene.
- **update: Instruction**: Instrucción que actualiza la variable de control o realiza alguna operación después de cada iteración.
- **instructions: Instruction[]**: Es una lista de instrucciones que se ejecutan en cada iteración del ciclo mientras la condición sea verdadera.

Constructor:

- **constructor(initialization: Instruction, condition: Expression, update: Instruction, instructions: Instruction[], line: number, column: number)**:

Inicializa una instancia de la clase For con las partes necesarias para ejecutar el ciclo (initialization, condition, update) y el conjunto de instrucciones a ejecutar. También incluye la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment)**

Este método ejecuta el ciclo for en el entorno proporcionado. Inicia con la ejecución de la instrucción de inicialización, luego evalúa la condición, y si es verdadera, ejecuta las instrucciones. Después de cada iteración, se ejecuta la actualización. El ciclo continúa hasta que la condición sea falsa o se encuentre una instrucción Break o Continue.

- **environment**: El entorno donde se ejecutará el ciclo y se evaluarán las expresiones.
- **Lanza**: Un error semántico si la condición no es de tipo booleana.

Proceso:

- **Crear un nuevo entorno**: Se crea un nuevo entorno para el ciclo for, separando el contexto de ejecución del ciclo del entorno externo.
- **Inicialización**: Ejecuta la instrucción de inicialización, que generalmente establece la variable de control del ciclo.
- **Evaluación de la condición**: Antes de cada iteración, se evalúa la condición. Si la condición es falsa, el ciclo se detiene.
- **Ejecución de instrucciones**: Si la condición es verdadera, se ejecutan las instrucciones dentro del ciclo.
- **Manejo de Break y Continue**: Si se encuentra un Break, el ciclo termina inmediatamente. Si se encuentra un Continue, salta a la siguiente iteración sin completar el resto de las instrucciones.
- **Actualización**: Al final de cada iteración, se ejecuta la instrucción de actualización, generalmente incrementando o modificando la variable de control.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa el ciclo for en un árbol de sintaxis abstracta (AST). Crea nodos para la inicialización, la condición, la actualización y las instrucciones del ciclo.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para el ciclo For.

Proceso:

- **Generar los nodos de las partes del ciclo**: Crea nodos para la inicialización, la condición y la actualización.
- **Crear el nodo principal**: Crea un nodo For que representa el ciclo.
- **Conectar los nodos**: Conecta los nodos de inicialización, condición y actualización al nodo principal.
- **Conectar las instrucciones**: Genera nodos para cada instrucción dentro del ciclo y los conecta al nodo principal.

Clase Funct

La clase Funct representa una función o un método dentro del entorno de ejecución. Las funciones pueden devolver un valor (dependiendo de su tipo de retorno), mientras que los métodos no devuelven ningún valor o están representados por void. Esta clase hereda de Instruction, lo que le permite ser ejecutada y registrada en el entorno de ejecución. A continuación, se describen los atributos y métodos principales.

Atributos:

- **id: string**: El identificador de la función o método, que actúa como su nombre.
- **tipoRetorno: DataType**: El tipo de dato que la función o método devuelve. Si se trata de un método, el tipo de retorno será VOID o NULO.
- **instructs: Statement**: El bloque de instrucciones o código que representa el cuerpo de la función o método.
- **parametros: { id: string, tipo: DataType, value: any }[]**: Lista de parámetros que la función o método acepta. Cada parámetro tiene un identificador, un tipo de dato y un valor por defecto.

Constructor:

- **constructor(id: string, tipoRetorno: DataType, instructs: Statement, parametros: { id: string, tipo: DataType, value: any }[], line: number, column: number)**:

Inicializa una instancia de la clase Funct con el identificador del método o función, el tipo de retorno, el bloque de código, los parámetros y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment)**

Este método guarda la función o método en el entorno global. Registra la función para que pueda ser llamada en el futuro mediante su identificador.

- **environment**: El entorno donde se registrará la función o método.

Proceso:

- **Obtener el entorno global**: Busca el entorno global desde el entorno actual.
- **Guardar la función o método**: Registra la función o método en el entorno global utilizando su identificador (id).

- **getParametros()**

Devuelve la lista de parámetros que acepta la función o método.

- **Retorna**: Un arreglo de objetos, donde cada objeto contiene el identificador, tipo y valor por defecto de un parámetro.

- **getInstrucciones()**

Devuelve el bloque de instrucciones (Statement) que constituye el cuerpo de la función o método.

- **Retorna**: Un objeto Statement que representa el bloque de código de la función o método.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la función o método en un árbol de sintaxis abstracta (AST). Crea nodos para los parámetros y el bloque de código.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la función o método.

Proceso:

- **Crear el nodo principal**: Representa la función o método con su nombre e indica si es una función o un método (dependiendo del tipo de retorno).
- **Generar nodos de parámetros**: Crea nodos para cada parámetro de la función o método, y los conecta al nodo principal.
- **Generar el nodo del cuerpo**: Crea un nodo para el bloque de código de la función y lo conecta al nodo principal.

Clase Increment

La clase Increment representa una instrucción que realiza la operación de incremento (sumar 1) sobre una variable en un entorno de ejecución. Esta clase hereda de Instruction, lo que le permite ser ejecutada dentro del flujo del programa y modificar el valor de una variable existente en el entorno. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **id: string**: Representa el identificador de la variable que se va a incrementar. Es el nombre de la variable que debe existir en el entorno.

Constructor:

- **constructor(id: string, line: number, column: number)**:

Inicializa una instancia de la clase Increment con el identificador de la variable y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método realiza la operación de incremento en la variable indicada. Busca la variable en el entorno, verifica que sea de tipo numérico, y suma 1 a su valor. Si la variable no existe o no es numérica, lanza un error semántico.

    - **environment**: El entorno donde se buscará la variable y se realizará el incremento.
    - **Retorna**: Un objeto Result con un valor nulo (NULO) porque es una operación de modificación de estado y no tiene un valor de retorno explícito.

- **Lanza**: Un error semántico si la variable no ha sido declarada o si el valor de la variable no es numérico.

Proceso:

- **Buscar la variable**: Verifica que la variable exista en el entorno. Si no existe, lanza un error.
- **Verificar el tipo de dato**: Asegura que el valor actual de la variable sea numérico. Si no lo es, lanza un error.
- **Incrementar el valor**: Suma 1 al valor actual de la variable.
- **Actualizar el entorno**: Almacena el nuevo valor en el entorno, actualizando la variable.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación de incremento en un árbol de sintaxis abstracta (AST). Crea un nodo para la variable que está siendo incrementada.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la operación de incremento.

Proceso:

- **Crear el nodo de incremento**: Representa la operación de incremento con el identificador de la variable.
- **Conectar nodos**: No hay nodos adicionales que conectar, ya que es una operación simple.

Clase Is

La clase Is es una expresión que verifica si el tipo de una expresión evaluada coincide con un tipo de dato específico. Esta clase hereda de Expression, lo que significa que puede ser utilizada dentro de otras expresiones y evaluada en un entorno de ejecución. Su propósito es realizar una verificación de tipos durante la ejecución del programa. A continuación, se describen sus atributos y métodos principales.

Atributos:

- **expression: Expression**: Es la expresión que será evaluada y cuyo tipo de dato será comparado.
- **tipo: DataType**: Es el tipo de dato con el cual se comparará el resultado de la evaluación de la expresión. Representa el tipo esperado.

Constructor:

- **constructor(expression: Expression, tipo: DataType, linea: number, columna: number)**:

Inicializa una instancia de la clase Is con la expresión a evaluar, el tipo con el que se realizará la comparación, y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método evalúa la expresión, obtiene su tipo de dato, y compara ese tipo con el tipo esperado. El resultado es un valor booleano que indica si el tipo coincide o no.

  - **environment**: El entorno donde se evaluará la expresión.
  - **Retorna**: Un objeto Result que contiene el valor booleano del resultado de la comparación (true si los tipos coinciden, false si no) y el tipo de dato (siempre BOOLEANO).

Proceso:

- **Evaluar la expresión**: Ejecuta la expresión proporcionada y obtiene su valor y tipo de dato.
- **Comparar el tipo**: Compara el tipo de la expresión con el tipo esperado almacenado en this.tipo.
- **Devolver el resultado**: Retorna un valor booleano (true o false) dependiendo de si los tipos coinciden.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la verificación de tipo en un árbol de sintaxis abstracta (AST). Crea nodos para la expresión evaluada y conecta estos nodos al nodo principal Is.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la expresión Is.

Proceso:

- **Crear el nodo de la expresión**: Representa la expresión que será evaluada.
- **Crear el nodo principal**: Representa la operación Is con el tipo de dato al que se compara la expresión.
- **Conectar los nodos**: Conecta el nodo de la expresión al nodo principal Is.

Clase Length

La clase Length representa una instrucción que calcula la longitud de una cadena, un arreglo, o un vector. Esta clase hereda de Instruction y evalúa una expresión en el entorno, verificando su tipo de dato y devolviendo su longitud. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **expression: Expression**: La expresión cuyo valor será evaluado para calcular su longitud. Puede ser una cadena, un arreglo o un vector.

Constructor:

- **constructor(expression: Expression, line: number, column: number)**:

Inicializa una instancia de la clase Length con la expresión a evaluar y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método evalúa la expresión proporcionada y calcula su longitud dependiendo de su tipo de dato. Si la expresión es una cadena, un vector (arreglo nativo), o una instancia de la clase Arreglo, retorna su longitud. Si no es de ninguno de estos tipos, lanza un error semántico.

  - **environment**: El entorno donde se evaluará la expresión.
  - **Retorna**: Un objeto Result con la longitud de la expresión evaluada y su tipo de dato (ENTERO).
  - **Lanza**: Un error semántico si la expresión no es de tipo cadena, vector o arreglo.

Proceso:

- **Evaluar la expresión**: Ejecuta la expresión proporcionada y obtiene su valor y tipo de dato.
- **Calcular la longitud**:
  - Si la expresión es una cadena (STRING), devuelve la longitud de la cadena.
  - Si la expresión es un arreglo nativo de JavaScript (Array), devuelve su longitud.
  - Si la expresión es una instancia de la clase Arreglo, utiliza su método getLength para obtener la longitud.
- **Manejo de errores**: Si la expresión no es de un tipo válido, lanza un error semántico indicando que el argumento no es compatible con la función len.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la operación de calcular la longitud en un árbol de sintaxis abstracta (AST). Crea un nodo para la expresión y lo conecta al nodo principal Length.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la operación Length.

Proceso:

- **Generar el nodo de la expresión**: Crea el nodo para la expresión cuya longitud se va a calcular.
- **Crear el nodo principal**: Representa la operación de calcular la longitud.
- **Conectar los nodos**: Conecta el nodo de la expresión al nodo principal Length.

Clase Lower

La clase Lower representa una instrucción que convierte una cadena de texto a minúsculas. Esta clase hereda de Instruction y evalúa una expresión de tipo cadena en el entorno de ejecución, retornando una versión en minúsculas del valor de la cadena.

Atributos:

- **expression: Expression**: Es la expresión que se evaluará y cuyo valor será convertido a minúsculas. Debe ser de tipo cadena (STRING).

Constructor:

- **constructor(expression: Expression, line: number, column: number)**: Inicializa una instancia de la clase Lower con la expresión a evaluar y la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**
  Este método evalúa la expresión proporcionada, verifica que sea de tipo cadena, y devuelve una versión de la cadena convertida a minúsculas.
  - **environment**: El entorno donde se evaluará la expresión.
  - **Retorna**: Un objeto Result con el valor de la cadena en minúsculas y su tipo de dato (STRING).
  - **Lanza**: Un error semántico si la expresión no es de tipo cadena.

Proceso:

- **Evaluar la expresión**: Ejecuta la expresión proporcionada y obtiene su valor y tipo de dato.
- **Verificar el tipo de dato**: Si el valor evaluado no es de tipo cadena (STRING), lanza un error semántico.

- **Convertir a minúsculas**: Si el valor es una cadena, lo convierte a minúsculas utilizando el método toLowerCase().
- **Devolver el resultado**: Retorna el valor convertido y el tipo de dato STRING.

- **generateNode(dotGenerator: DotGenerator): string**
  Genera un nodo gráfico en formato DOT que representa la operación de conversión a minúsculas en un árbol de sintaxis abstracta (AST). Crea un nodo para la expresión evaluada y lo conecta al nodo principal Lower.
  - **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
  - **Retorna**: Un identificador en formato string que representa el nodo creado para la operación Lower.

Proceso:

- **Generar el nodo de la expresión**: Crea el nodo para la expresión que se convertirá a minúsculas.
- **Crear el nodo principal**: Representa la operación de conversión a minúsculas.
- **Conectar los nodos**: Conecta el nodo de la expresión al nodo principal Lower.

Clase MatrixAssignment

La clase MatrixAssignment representa una instrucción de asignación de valor en una matriz, donde se especifica una fila y una columna dentro de la matriz para asignar un valor. Esta clase hereda de Instruction, lo que permite ejecutarla dentro del flujo del programa y modificar una matriz existente en el entorno. A continuación, se describen los atributos y métodos principales de la clase.

Atributos:

- **id: string**: El identificador de la matriz en el entorno, que es el nombre de la variable que debe contener la matriz.
- **row: Expression**: La expresión que determina la fila de la matriz donde se realizará la asignación.
- **colExpr: Expression**: La expresión que determina la columna de la matriz donde se realizará la asignación.
- **value: Expression**: El valor que se asignará en la posición especificada de la matriz.

Constructor:

- **constructor(id: string, row: Expression, colExpr: Expression, value: Expression, line: number, column: number)**:

Inicializa una instancia de la clase MatrixAssignment con los valores de fila, columna y el valor que se asignará en la matriz, junto con la ubicación en el código fuente (línea y columna) para manejo de errores y depuración.

Métodos:

- **execute(environment: Environment): Result**

Este método ejecuta la asignación en la matriz. Evalúa las expresiones de fila, columna y valor, verifica que sean correctos y que coincidan con los tipos de la matriz, y luego realiza la asignación en la posición especificada.

- **environment**: El entorno donde se realizará la asignación.
- **Retorna**: Un objeto Result con un valor nulo (NULO) ya que no se devuelve un valor útil en una asignación.
- **Lanza**: Un error semántico si:
  - La matriz no existe.
  - Los índices de fila o columna no son números.
  - El tipo de dato del valor no coincide con el tipo de la matriz.

Proceso:

- **Buscar la matriz**: Verifica que la variable especificada por el identificador sea una matriz.
- **Evaluar los índices**: Evalúa las expresiones que indican las posiciones de fila y columna.
- **Verificar los tipos**: Asegura que los índices sean números y que el valor a asignar coincida con el tipo de la matriz.
- **Asignar el valor**: Si todo es correcto, asigna el valor en la posición [fila][columna] de la matriz.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la asignación en una matriz en un árbol de sintaxis abstracta (AST). Crea nodos para las expresiones de fila, columna y valor, y los conecta al nodo principal de la asignación.

- **dotGenerator**: Instancia de DotGenerator utilizada para generar el nodo en el AST.
- **Retorna**: Un identificador en formato string que representa el nodo creado para la operación de asignación en la matriz.

Proceso:

- **Generar nodos de fila, columna y valor**: Representa las expresiones de fila, columna y el valor a asignar.
- **Crear el nodo principal**: Representa la operación de asignación en la matriz.
- **Conectar los nodos**: Conecta los nodos de fila, columna y valor al nodo principal.

Clase MatrixDeclaration

La clase MatrixDeclaration representa la declaración de matrices en un entorno de ejecución. Se pueden declarar matrices con un tamaño específico o con valores predefinidos. Esta clase hereda de Instruction, lo que le permite integrarse en el flujo de instrucciones del programa. A continuación, se describen los atributos y métodos principales.

Atributos:

- **tipo: DataType**: El tipo de datos que almacenará la matriz (ENTERO, STRING, BOOLEANO, etc.).
- **ids: string[]**: Lista de identificadores (nombres) de las matrices que se declararán.
- **filas: Expression | null**: Número de filas de la matriz, expresado como una expresión (para el tipo de declaración basada en tamaño).
- **columnas: Expression | null**: Número de columnas de la matriz, expresado como una expresión (para el tipo de declaración basada en tamaño).
- **values: Expression[][] | null**: Valores predefinidos para la matriz, organizados en filas y columnas (para el tipo de declaración basada en valores).

Constructor:

- **constructor(tipo: DataType, ids: string[], filas: Expression | null, columnas: Expression | null, values: Expression[][] | null, line: number, column: number)**

Inicializa una instancia de la clase MatrixDeclaration con los datos sobre el tipo de la matriz, los identificadores, las dimensiones o los valores predefinidos, y la ubicación en el código fuente.

Métodos:

- **execute(environment: Environment): Result**

Ejecuta la declaración de la matriz en el entorno. Dependiendo de los parámetros proporcionados, declara la matriz con un tamaño específico o con valores predefinidos.

- **environment**: El entorno donde se realizará la declaración de la matriz.
- **Retorna**: Un objeto Result con un valor NULO, ya que la declaración no genera un valor.
- **Lanza**: Un error semántico si:
  - No se proporcionan valores o dimensiones válidas.
  - El tipo de dato de los valores no coincide con el tipo de la matriz.

Tipos de declaración:

- **Declaración con valores predefinidos**:
  - Evalúa las expresiones que representan los valores de la matriz y verifica que coincidan con el tipo de la matriz.
  - Declara la matriz y la guarda en el entorno con los valores evaluados.
- **Declaración con tamaño**:
  - Evalúa las expresiones para filas y columnas.
  - Crea una matriz con valores por defecto según el tipo especificado y la guarda en el entorno.

- **declareWithValues(environment: Environment)**

Este método se encarga de la declaración de la matriz cuando se proporcionan valores predefinidos. Evalúa cada valor en la matriz y la declara en el entorno.

- **Lanza**: Un error si los tipos de los valores no coinciden con el tipo de la matriz.

- **declareWithSize(environment: Environment)**

Este método maneja la declaración de la matriz basada en el tamaño especificado. Evalúa las expresiones para filas y columnas y luego crea una matriz con valores por defecto.

- **Lanza**: Un error si las dimensiones no son válidas (por ejemplo, si las filas o columnas no son números o son menores o iguales a cero).

- **getDefaultValueByType(tipo: DataType): any**

Retorna el valor por defecto según el tipo de dato especificado (ENTERO, DECIMAL, STRING, etc.).

  - **tipo**: El tipo de dato para el cual se quiere obtener el valor por defecto.
  - **Retorna**: El valor por defecto para el tipo proporcionado.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT que representa la declaración de la matriz en un árbol de sintaxis abstracta (AST).

  - **dotGenerator**: Instancia de DotGenerator para generar el nodo.
  - **Retorna**: Un identificador en formato string que representa el nodo generado para la declaración de la matriz.

Proceso:

  - **Declaración con tamaño**: Crea nodos para las dimensiones de la matriz (filas y columnas) y los conecta al nodo principal.
  - **Declaración con valores**: Crea nodos para cada valor de la matriz y los conecta al nodo principal.

Clase Max

La clase Max es una expresión que se utiliza para encontrar el valor máximo en un vector o arreglo. Hereda de la clase Expression, lo que la convierte en una expresión evaluable dentro de un entorno. El valor máximo depende del tipo de datos almacenados en el vector, ya sea numérico, carácter, booleano o cadena.

Atributos:

- **id: string**: El identificador del vector o arreglo del cual se calculará el valor máximo.

Constructor:

- **constructor(id: string, line: number, column: number)**

Inicializa una instancia de la clase Max con el identificador del arreglo, la línea y la columna en las que se encuentra la expresión en el código fuente.

Métodos:

- **execute(environment: Environment): Result**

Este método evalúa el vector en el entorno actual y calcula el valor máximo en función del tipo de datos almacenados en el vector. Dependiendo del tipo de datos, el valor máximo se calcula de la siguiente manera:

  - **Numeros (ENTERO, DECIMAL)**: Se utiliza Math.max para encontrar el mayor número.
  - **Caracteres (CHAR)**: Se compara el valor ASCII de los caracteres.
  - **Booleanos (BOOLEANO)**: true se considera mayor que false.
  - **Cadenas (STRING)**: Se realiza una comparación lexicográfica.

Si el tipo de datos no es compatible con la operación max, se lanza un error semántico.

  - **environment**: El entorno donde se encuentra el arreglo.

    - **Retorna**: Un objeto Result que contiene el valor máximo encontrado y su tipo de dato.
    - **Lanza**: Un error semántico si el identificador no corresponde a un arreglo o si el tipo de dato no es compatible con la operación max.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo gráfico en formato DOT para representar la operación Max en un árbol de sintaxis abstracta (AST).

    - **dotGenerator**: Instancia de DotGenerator para crear el nodo.
    - **Retorna**: Un identificador en formato string que representa el nodo generado para la expresión Max.

Clase MethodCall

La clase MethodCall representa la ejecución de una subrutina (función o método) dentro del entorno de ejecución. Hereda de la clase Expression, lo que indica que su resultado es evaluable y puede devolver un valor (como una función).

Atributos:

- **id: string**: El identificador de la subrutina (función o método) que se va a ejecutar.
- **args: { id: string, value: any }[]**: Lista de argumentos pasados a la subrutina, cada uno con un identificador y un valor.

Constructor:

- **constructor(id: string, args: { id: string, value: any }[], line: number, column: number)**

Inicializa una instancia de la clase MethodCall con el identificador de la subrutina, los argumentos y la ubicación en el código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result**

Este método busca la subrutina en el entorno actual y la ejecuta. Asigna los valores de los argumentos a los parámetros de la subrutina en un nuevo entorno. Dependiendo de si la subrutina es un método (void) o una función, devuelve un resultado o null.

    - **environment**: El entorno donde se ejecutará la subrutina.
    - **Retorna**: Un objeto Result con el valor y tipo de dato resultante de la ejecución de la subrutina. Si es un método (void), devuelve null.
    - **Lanza**: Un error semántico si la subrutina no está definida, si los argumentos no coinciden con los parámetros, o si falta algún valor para un parámetro.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT para representar la llamada a la subrutina en el árbol de sintaxis abstracta (AST).

    - **dotGenerator**: El generador de nodos DOT para visualización gráfica.
    - **Retorna**: Un identificador en formato string que representa el nodo generado para la llamada a la subrutina.

Clase Min

La clase Min representa una expresión que calcula el valor mínimo de un vector o arreglo en el entorno de ejecución. Hereda de la clase Expression, lo que permite evaluarla y obtener un resultado durante la ejecución.

Atributos:

- **id: string**: El identificador del vector o arreglo del cual se calculará el valor mínimo.

Constructor:

- **constructor(id: string, line: number, column: number)**

Inicializa una instancia de la clase Min con el identificador del vector y la ubicación en el código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result**

Este método obtiene el vector del entorno, verifica que sea válido, y calcula el valor mínimo. El tipo de dato del vector puede ser ENTERO, DECIMAL, CHAR, BOOLEANO o STRING. Dependiendo del tipo, realiza la comparación adecuada para encontrar el valor mínimo.

    - **environment**: El entorno donde se encuentra el vector.
    - **Retorna**: Un objeto Result con el valor mínimo encontrado y su tipo de dato.
    - **Lanza**: Un error semántico si el identificador no es un vector válido o si el tipo de datos del vector no es compatible con la función min.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT para representar la operación Min en el árbol de sintaxis abstracta (AST).

- **dotGenerator**: El generador de nodos DOT para visualización gráfica.
- **Retorna**: Un identificador en formato string que representa el nodo generado para la operación Min.

Clase Return

La clase Return representa una instrucción de retorno dentro de una función o método en el entorno de ejecución. Permite retornar un valor específico o simplemente indicar el final de la ejecución de la función sin devolver un valor explícito.

Atributos:

- **expression: Expression | null**: La expresión opcional que se evalúa y se devuelve como resultado de la función. Puede ser nula si no hay expresión de retorno.

Constructor:

- **constructor(expression: Expression | null, line: number, column: number)**

Inicializa una instancia de la clase Return con una expresión opcional y la ubicación de la instrucción en el código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result | null**

Este método ejecuta la instrucción de retorno. Si hay una expresión, la evalúa y retorna el valor resultante. Si no hay expresión, retorna un valor null con el tipo DataType.RETURN.

- **environment**: El entorno de ejecución actual donde se evalúa la expresión.

    - **Retorna**: Un objeto Result con el valor de la expresión y su tipo de dato, o un Result con valor null y tipo DataType.RETURN si no hay expresión de retorno.
- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT para representar la instrucción Return en el árbol de sintaxis abstracta (AST).

    - **dotGenerator**: El generador de nodos DOT para visualización gráfica.
    - **Retorna**: Un identificador en formato string que representa el nodo generado para la instrucción Return. Si hay una expresión asociada, también genera y conecta su nodo correspondiente.

Clase Reverse

La clase Reverse representa una instrucción que invierte los valores de un arreglo en el entorno de ejecución. Esta instrucción actúa sobre un vector identificado por su nombre y actualiza sus valores con el orden invertido.

Atributos:

- **id: string**: Identificador del arreglo cuyo contenido será invertido.

Constructor:

- **constructor(id: string, line: number, column: number)**

Inicializa una instancia de la clase Reverse con el identificador del arreglo y la ubicación de la instrucción en el código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result**

Este método ejecuta la inversión del arreglo. Primero busca el arreglo en el entorno, luego invierte sus valores y los actualiza en el entorno.

  - **environment**: El entorno de ejecución donde se encuentra el arreglo.
  - **Retorna**: Un objeto Result que contiene el arreglo invertido y su tipo de dato (DataType.ARRAY).
  - **Proceso de ejecución**:
    - Busca el arreglo en el entorno por su identificador.
    - Verifica que la variable encontrada sea un arreglo válido.
    - Invierte el contenido del arreglo según su tipo de datos.
    - Actualiza el arreglo con los valores invertidos.
    - Devuelve el arreglo invertido como resultado.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT para representar la instrucción Reverse en el árbol de sintaxis abstracta (AST).

  - **dotGenerator**: El generador de nodos DOT para visualización gráfica.
  - **Retorna**: Un identificador en formato string que representa el nodo generado para la instrucción Reverse.

Clase Statement

La clase Statement representa un bloque de instrucciones que se ejecutan secuencialmente en un entorno de ejecución. Este tipo de bloque es común en funciones, ciclos, o bloques condicionales, donde se agrupan varias instrucciones para ejecutarlas de manera continua en un mismo contexto (entorno). La clase reutiliza el entorno que se le pasa, en lugar de crear uno nuevo, lo que permite que los cambios en las variables sean persistentes dentro de dicho entorno.

Atributos:

- **code: Instruction[]**: Un arreglo de instrucciones que conforman el bloque de código.
- **line: number**: Línea del código donde se define el bloque, útil para la depuración.
- **column: number**: Columna del código donde se define el bloque.

Constructor:

- **constructor(code: Instruction[], line: number, column: number)**

Inicializa la clase con un conjunto de instrucciones (code) y la ubicación en el código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result | undefined**

Ejecuta secuencialmente todas las instrucciones del bloque en el entorno actual. Si alguna instrucción devuelve un resultado (como un retorno en una función), se detiene la ejecución del bloque y se devuelve dicho resultado.

    - **environment**: El entorno donde se ejecutarán las instrucciones.
    - **Retorna**: El resultado de una instrucción que lo requiera (como un Return, Break, o Continue), o undefined si ninguna instrucción devuelve un resultado significativo.

- **Proceso de ejecución**:
    - Itera sobre cada instrucción en el bloque.
    - Ejecuta cada instrucción en el entorno.
    - Si se encuentra una instrucción Return, se devuelve el valor resultante.
    - Si se encuentra una instrucción Break o Continue, se detiene el ciclo y se devuelve la instrucción encontrada.
    - Captura cualquier error que ocurra durante la ejecución de una instrucción.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo en formato DOT para representar gráficamente el bloque de instrucciones.

- **dotGenerator**: El generador de nodos DOT para visualización gráfica.
- **Retorna**: Un identificador de nodo en formato string que representa el bloque Statement en el árbol de sintaxis abstracta (AST).

Clase Sum

La clase Sum permite calcular la suma o concatenación de los elementos de un arreglo en función de su tipo de datos. Es una expresión que puede ser evaluada en tiempo de ejecución para obtener un valor resultante que depende del tipo de los elementos almacenados en el arreglo.

Atributos:

- **id: string**: El identificador del arreglo cuyo contenido será sumado o concatenado.
- **line: number**: Línea del código fuente donde se encuentra la expresión, útil para la depuración y manejo de errores.
- **column: number**: Columna del código fuente donde se encuentra la expresión.

Constructor:

- **constructor(id: string, line: number, column: number)**

Inicializa la expresión Sum con el identificador del arreglo y la posición en el código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result**

Este método realiza la suma o concatenación de los elementos de un arreglo dependiendo de su tipo de datos. Lanza un error si el arreglo no existe o no es de un tipo válido para la operación.

    - **Proceso de ejecución**:
        - Busca el arreglo en el entorno.
        - Verifica que la variable existe y es un arreglo.

        - Dependiendo del tipo de datos del arreglo, realiza una de las siguientes operaciones:
            - **ENTERO/DECIMAL**: Suma de valores numéricos.
            - **CHAR**: Suma de los valores ASCII de los caracteres.
            - **BOOLEANO**: Suma los valores booleanos considerando true como 1 y false como 0.
            - **STRING**: Concatenación de todas las cadenas.
        - Retorna el valor sumado o concatenado junto con su tipo de datos.
    - **Retorna**: Un objeto Result que contiene el valor sumado/concatenado y el tipo de datos resultante.
- **generateNode(dotGenerator: DotGenerator): string**

Este método genera un nodo en formato DOT para visualizar el árbol de sintaxis abstracta (AST).

    - **dotGenerator**: El generador de nodos DOT para visualización gráfica.
    - **Retorna**: Un identificador de nodo en formato string que representa la operación Sum en el AST.

Clase Switch

La clase Switch permite la evaluación de una expresión y la ejecución condicional de un bloque de instrucciones basado en los casos definidos. Cada caso se compara con el valor de la expresión proporcionada, y si hay coincidencia, se ejecutan las instrucciones correspondientes. También permite manejar un bloque default en caso de que ningún caso coincida.

Atributos:

- **expression: Expression**: La expresión que se evaluará en el switch para comparar con los casos.
- **cases: Case[]**: Un arreglo de bloques Case que contienen las expresiones y las instrucciones que se ejecutarán si hay coincidencia con el valor del switch.
- **defaultCase: Default | null**: Un bloque Default opcional que se ejecutará si ninguno de los Case coincide con la expresión del switch.

Constructor:

- **constructor(expression: Expression, cases: Case[], defaultCase: Default | null, line: number, column: number)**

Inicializa la instrucción Switch con la expresión de evaluación, los bloques de Case, un bloque Default opcional, y las coordenadas del código fuente.

Métodos:

- **execute(environment: Environment): any**

Este método ejecuta el switch, evaluando la expresión principal y comparándola con los valores de los casos. Si encuentra una coincidencia, ejecuta las instrucciones del bloque Case correspondiente. Si un bloque Case coincide, continuará ejecutando los casos subsiguientes hasta encontrar un break. Si ningún caso coincide y hay un bloque Default, se ejecuta este bloque.

    - **Proceso de ejecución**:

        - Evalúa la expresión del switch.
        - Itera sobre los casos, comparando los valores de cada Case con el valor del switch.
        - Si se encuentra una coincidencia, se ejecuta el bloque del Case y se verifica si hay un break.
        - Si no se encuentra coincidencia y existe un bloque Default, se ejecuta dicho bloque.
        - El ciclo se interrumpe si encuentra un break.
- **generateNode(dotGenerator: DotGenerator): string**

Genera el nodo en formato DOT para representar el árbol de sintaxis abstracta (AST).

    - **dotGenerator**: Generador de nodos DOT para la representación gráfica.
    - **Retorna**: Un identificador de nodo en formato string que representa el Switch en el AST.

Clase ToCharArray

La clase ToCharArray convierte una cadena de texto (string) en un arreglo de caracteres (char[]). Se trata de una instrucción que toma una expresión de tipo string, la convierte en un arreglo de caracteres y devuelve el resultado como un objeto de tipo Arreglo.

Atributos:

- **expression: Expression**: La expresión que se evaluará para convertirla en un arreglo de caracteres.

Constructor:

- **constructor(expression: Expression, line: number, column: number)**

Inicializa la instrucción ToCharArray con una expresión de tipo string y las coordenadas del código fuente (línea y columna).

Métodos:

- **execute(environment: Environment): Result**

Este método evalúa la expresión, verifica que sea de tipo string, y la convierte en un arreglo de caracteres. Si la evaluación no es de tipo string, lanza un error semántico.

    - **Proceso de ejecución**:
        - Evalúa la expresión y obtiene su valor.
        - Verifica que el tipo de dato sea DataType.STRING.
        - Si es válido, convierte la cadena en un arreglo de caracteres.
        - Crea un objeto Arreglo de tipo CHAR con los caracteres.
        - Retorna el arreglo como resultado con tipo ARRAY.
- **generateNode(dotGenerator: DotGenerator): string**

Genera el nodo DOT para representar gráficamente esta instrucción en el árbol de sintaxis abstracta (AST).

    - **dotGenerator**: Generador de nodos DOT para la representación gráfica.
    - **Retorna**: Un identificador de nodo en formato string que representa la instrucción ToCharArray en el AST.

Clase ToString

La clase ToString convierte una expresión de tipo numérico (int o double) o booleano (boolean) a una cadena de texto (string). Se trata de una instrucción que evalúa una expresión y devuelve su representación en forma de texto.

Atributos:

- **expression: Expression**: La expresión que será evaluada y convertida a cadena.

Constructor:

- **constructor(expression: Expression, line: number, column: number)**

Inicializa la instrucción ToString con una expresión que debe ser de tipo numérico o booleano, junto con las coordenadas de línea y columna en el código fuente.

Métodos:

- **execute(environment: Environment): Result**

Este método evalúa la expresión, verifica que sea de tipo numérico (int, double) o booleano, y luego convierte el valor de la expresión a una cadena de texto. Si la expresión no es de uno de los tipos permitidos, lanza un error semántico.

    - **Proceso de ejecución**:
        - Evalúa la expresión y obtiene su valor.
        - Verifica si el tipo de dato es ENTERO, DECIMAL o BOOLEANO.
        - Si es válido, convierte el valor a su representación en cadena usando el método .toString().
        - Retorna el valor convertido como resultado con tipo STRING.
- **generateNode(dotGenerator: DotGenerator): string**

Genera el nodo DOT para representar gráficamente esta instrucción en el árbol de sintaxis abstracta (AST).

    - **dotGenerator**: Generador de nodos DOT para la representación gráfica.
    - **Retorna**: Un identificador de nodo en formato string que representa la instrucción ToString en el AST.

Clases Break y Continue

Las clases Break y Continue representan instrucciones de control de flujo para la interrupción y continuación en ciclos de control como for, while, o switch. Ambas clases heredan de la clase base Instruction y son instrucciones especiales que no devuelven un valor, sino que retornan a sí mismas para ser manejadas por la estructura de control correspondiente.

Clase Break

La clase Break permite interrumpir la ejecución de un ciclo o estructura condicional (como for, while, o switch). Cuando es ejecutada, señala al ciclo que debe detener su ejecución inmediatamente.

Atributos:

- **Línea** y **columna**: La posición en el código donde se encuentra la instrucción Break.

Constructor:

- **constructor(line: number, column: number)**

Inicializa la instrucción Break con la línea y la columna de código donde se utiliza.

Métodos:

- **execute(): Break**

Simplemente retorna la propia instancia para que las estructuras de control (switch, for, while, etc.) puedan manejarla y detener la ejecución del bloque.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo DOT para representar gráficamente la instrucción Break en el árbol de sintaxis abstracta (AST).

    - **Retorna**: Un identificador de nodo en formato string que representa la instrucción Break.

Clase Continue

La clase Continue se utiliza para continuar con la siguiente iteración de un ciclo (for, while). Cuando es ejecutada, señala al ciclo que debe omitir el resto de la iteración actual y pasar directamente a la siguiente.

Atributos:

- **Línea** y **columna**: La posición en el código donde se encuentra la instrucción Continue.

Constructor:

- **constructor(line: number, column: number)**

Inicializa la instrucción Continue con la línea y la columna de código donde se utiliza.

Métodos:

- **execute(): Continue**

Retorna la propia instancia de Continue, que es manejada por un ciclo para saltar a la siguiente iteración.

- **generateNode(dotGenerator: DotGenerator): string**

Genera un nodo DOT para representar gráficamente la instrucción Continue en el AST.

  - **Retorna**: Un identificador de nodo en formato string que representa la instrucción Continue.

Clase Upper

La clase Upper representa una instrucción que convierte una cadena de texto a mayúsculas. Esta instrucción toma una expresión de tipo string, evalúa su valor y convierte todas sus letras a mayúsculas.

Atributos:

- **expression: Expression**
- La expresión que se evaluará y cuyo valor será convertido a mayúsculas.

Constructor:

- **constructor(expression: Expression, line: number, column: number)**

Inicializa la clase con la expresión que será evaluada, además de la línea y la columna donde está presente en el código.

Métodos:

execute(environment: Environment): Result

- **Descripción**:

Este método evalúa la expresión dada y verifica si su tipo es string. Si lo es, convierte la cadena a mayúsculas y devuelve un resultado con el valor convertido. Si no es de tipo string, lanza un error semántico.

- **Parámetros**:
  - environment: El entorno de ejecución actual.
- **Retorna**:

Un objeto Result que contiene la cadena convertida a mayúsculas y el tipo DataType.STRING.

- **Lanza**:

Un error semántico si el tipo de la expresión no es una cadena.

generateNode(dotGenerator: DotGenerator): string

- **Descripción**:

Genera un nodo DOT para la visualización en el árbol de sintaxis abstracta (AST). Este nodo representa la conversión a mayúsculas.

- **Parámetros**:
  - dotGenerator: Instancia de DotGenerator que maneja la creación de nodos y conexiones en el gráfico.

- **Retorna**:

El nombre del nodo generado en formato string.

Clase VectorAssignment

La clase VectorAssignment representa una instrucción que permite asignar un valor a una posición específica dentro de un vector unidimensional. Esta instrucción toma el identificador del vector, el índice de la posición donde se va a asignar el valor, y el valor que se asignará.

Atributos:

- **id: string**
- El nombre o identificador del vector.
- **index1: Expression**

La expresión que define el índice de la posición en el vector a la que se asignará el valor.

- **value: Expression**

La expresión cuyo valor será evaluado y asignado en el índice del vector.

Constructor:

- **constructor(id: string, index1: Expression, value: Expression, linea: number, columna: number)**

Inicializa la clase con el identificador del vector, la expresión que define el índice, la expresión para el valor a asignar, y la información de línea y columna en el código.

Métodos:

execute(environment: Environment)

- **Descripción**:

Ejecuta la asignación en el vector. Busca el vector en el entorno, evalúa el índice y el valor a asignar, valida los tipos de datos, y realiza la asignación si todo es correcto.

- **Parámetros**:
  - environment: El entorno de ejecución donde se buscará el vector y se evaluarán las expresiones.
- **Lanza**:
  - Un error semántico si:
    - El vector no ha sido declarado.
    - El índice no es un número.
    - El índice está fuera de los límites del vector.
    - El tipo de dato del valor no coincide con el tipo del vector.

generateNode(dotGenerator: DotGenerator): string

- **Descripción**:

Genera un nodo DOT para la representación del árbol de sintaxis abstracta (AST), representando la asignación en el vector.

- **Parámetros**:
  - dotGenerator: Instancia de DotGenerator que maneja la creación de nodos y conexiones para la visualización del AST.
- **Retorna**:

El nombre del nodo generado en formato string.

Clase VectorDeclaration

La clase VectorDeclaration representa la declaración de un vector unidimensional en un entorno dado. Permite declarar un vector tanto con un tamaño específico como con una lista de valores predefinidos.

Atributos:

- **tipo: DataType**
- El tipo de datos que contendrá el vector (ENTERO, DECIMAL, STRING, etc.).
- **ids: string[]**

Lista de identificadores para los vectores que se van a declarar.

- **size1: Expression | null**

Expresión que define el tamaño del vector (usada para la declaración de tamaño).

- **values: Expression[] | null**

Lista de valores predefinidos para la declaración del vector (usada para la declaración con valores).

Constructor:

- **constructor(tipo: DataType, ids: string[], size1: Expression | null, values: Expression[] | null, line: number, column: number)**

Inicializa la clase con el tipo de datos del vector, los identificadores, la expresión del tamaño (si corresponde) o los valores predefinidos, y la información de línea y columna en el código.

Métodos:

execute(environment: Environment): Result

- **Descripción**:

Ejecuta la declaración del vector en el entorno. Dependiendo de si se especificó un tamaño o una lista de valores, se procederá con la declaración correspondiente.

- **Parámetros**:
  - environment: El entorno de ejecución donde se guardará el vector.
- **Lanza**:
  - Un error semántico si:
    - No se proporcionan ni un tamaño ni valores.
    - El tipo de dato de los valores no coincide con el tipo especificado para el vector.

declareWithValues(environment: Environment)

- **Descripción**:

Maneja la declaración del vector con una lista de valores predefinidos.

- **Parámetros**:
  - environment: El entorno de ejecución donde se guardará el vector.
- **Lanza**:
  - Un error semántico si el tipo de los valores no coincide con el tipo del vector.

declareWithSize(environment: Environment)

- **Descripción**:

Maneja la declaración del vector con un tamaño especificado. Inicializa el vector con valores por defecto según el tipo de dato.

- **Parámetros**:
  - environment: El entorno de ejecución donde se guardará el vector.

- **Lanza**:
    - Un error semántico si el tamaño del vector no es válido.

getDefaultValueByType(tipo: DataType): any

- **Descripción**:

Devuelve el valor por defecto correspondiente al tipo de datos del vector.

- **Parámetros**:
    - tipo: El tipo de datos del vector.
- **Retorna**:

El valor por defecto para el tipo de datos especificado.

generateNode(dotGenerator: DotGenerator): string

- **Descripción**:

Genera un nodo DOT para la visualización del árbol de sintaxis abstracta (AST), representando la declaración del vector.

- **Parámetros**:
    - dotGenerator: Instancia de DotGenerator que maneja la creación de nodos y conexiones para la visualización del AST.
- **Retorna**:

El nombre del nodo generado en formato string.

Clase While

La clase While representa la instrucción de un ciclo while en un entorno de ejecución. Este ciclo ejecuta repetidamente un bloque de código mientras la condición evaluada sea verdadera.

Atributos:

- **condition: Expression**
- La expresión booleana que determina si el ciclo continuará o se detendrá.
- **statementBlock: Statement**

El bloque de instrucciones (representado por un Statement) que se ejecuta en cada iteración del ciclo mientras la condición sea verdadera.

Constructor:

- **constructor(condition: Expression, statementBlock: Statement, line: number, column: number)**

Inicializa la clase con la condición del ciclo, el bloque de instrucciones que se ejecutará, y la información de línea y columna en el código.

Métodos:

execute(environment: Environment): void

- **Descripción**:

Ejecuta el ciclo while. Evalúa la condición antes de cada iteración y ejecuta el bloque de instrucciones mientras la condición sea verdadera. También maneja las sentencias Break y Continue para controlar la ejecución del ciclo.

- **Parámetros**:
  - environment: El entorno de ejecución en el que se evalúa la condición y se ejecuta el bloque de instrucciones.
- **Lanza**:
  - Un error semántico si la condición no es booleana.

generateNode(dotGenerator: DotGenerator): string

- **Descripción**:

Genera un nodo DOT para la visualización del árbol de sintaxis abstracta (AST), representando la estructura del ciclo while.

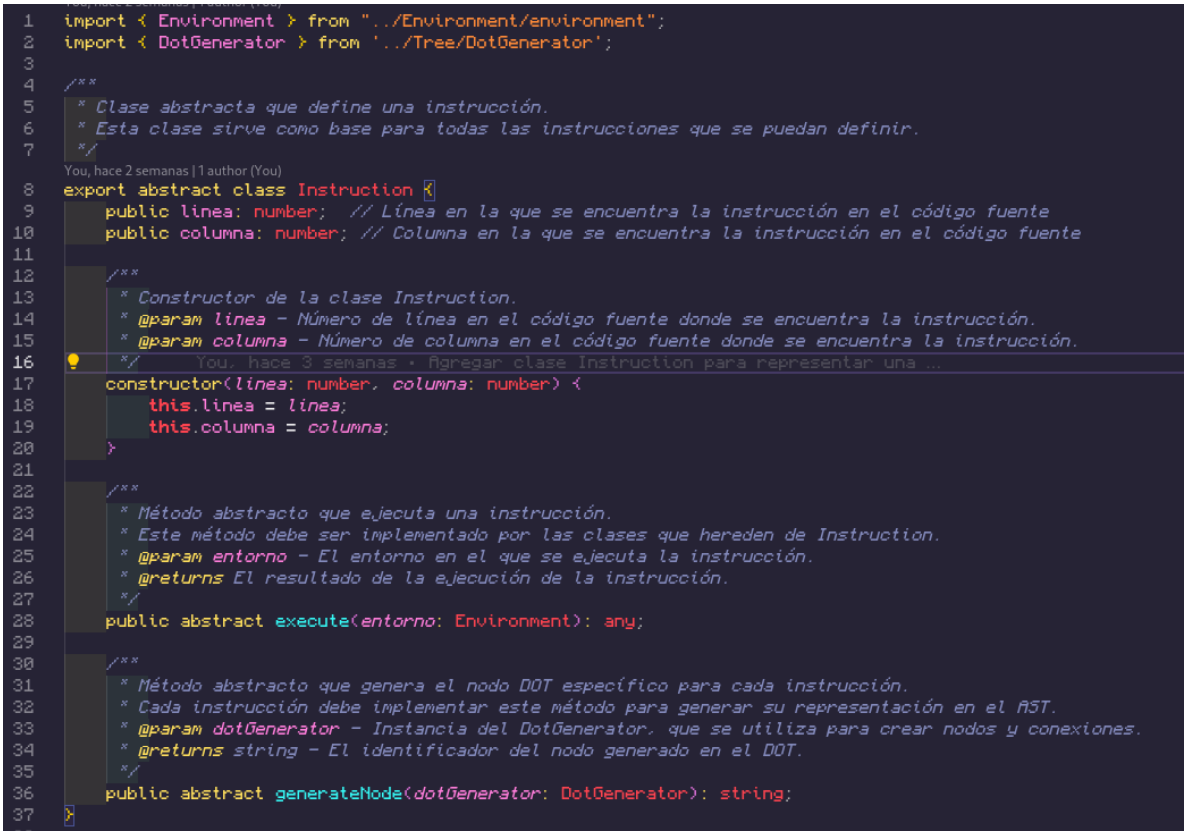
- **Parámetros**:
  - dotGenerator: Instancia de DotGenerator que maneja la creación de nodos y conexiones para la visualización del AST.
- **Retorna**:

El nombre del nodo generado en formato string.

```typescript
import { Environment } from "../Environment/environment";
import { Instruction } from "../abstract/instruction";
import { DotGenerator } from './DotGenerator'; // Importa tu DotGenerator
import { Execute } from "../Instructions/execute";
let consola: string[] = [];

You, ayer | 1 author (You)
export class AST {

    private globalEnv: Environment | null = null;
    private dotGenerator: DotGenerator; // Instancia de DotGenerator

    constructor(private instructions: Instruction[]) {
        this.dotGenerator = new DotGenerator(); // Inicializa el DotGenerator
    }

    interpreter(): string[] {
        this.globalEnv = new Environment(null, 'Global'); // Crea el entorno global
        consola = []; // Reinicia la consola

        // Fase 1: Registrar todas las funciones (y otras instrucciones necesarias) en el entorno global
        for (const instruction of this.instructions) {
            try {
                if (!(instruction instanceof Execute)) { // Evitar volver a registrar funciones
                    instruction.execute(this.globalEnv); // Registrar la función
                }
            } catch (error) {
                console.error(`Error al registrar función: ${error}`);
            }
        }

        // Fase 2: Ejecutar el resto de las instrucciones
        for (const instruction of this.instructions) {
            try {      You, hace 6 días • Refactorizar componente Access para agregar un ...
                if (instruction instanceof Execute) { // Si es una definición de función
                    instruction.execute(this.globalEnv); // Ejecutar las otras instrucciones
                }
            } catch (error) {
                console.error(error); // Captura y muestra errores
            }
        }

        return consola; // Retorna la consola con los resultados de la ejecución
    }

```

Figura 1. Definición de la clase AST.

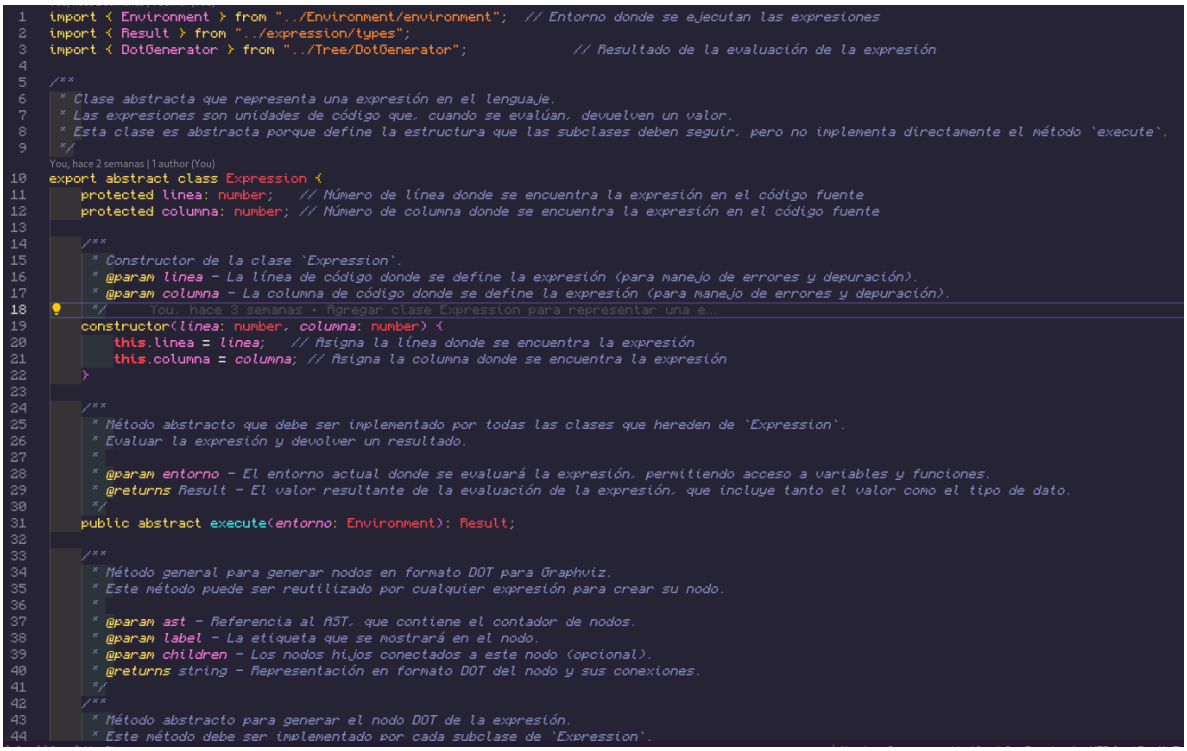
Fuente: Elaboración propia.



```
import { Environment } from "../Environment/environment";
import { DotGenerator } from '../Tree/DotGenerator';

/**
 * Clase abstracta que define una instrucción.
 * Esta clase sirve como base para todas las instrucciones que se puedan definir.
 */
You, hace 2 semanas | 1 author (You)
export abstract class Instruction {
    public linea: number;  // Línea en la que se encuentra la instrucción en el código fuente
    public columna: number; // Columna en la que se encuentra la instrucción en el código fuente

    /**
     * Constructor de la clase Instruction.
     * @param linea - Número de línea en el código fuente donde se encuentra la instrucción.
     * @param columna - Número de columna en el código fuente donde se encuentra la instrucción.
     */     You, hace 3 semanas • Agregar clase Instruction para representar una ...
    constructor(linea: number, columna: number) {
        this.linea = linea;
        this.columna = columna;
    }

    /**
     * Método abstracto que ejecuta una instrucción.
     * Este método debe ser implementado por las clases que hereden de Instruction.
     * @param entorno - El entorno en el que se ejecuta la instrucción.
     * @returns El resultado de la ejecución de la instrucción.
     */
    public abstract execute(entorno: Environment): any;

    /**
     * Método abstracto que genera el nodo DOT específico para cada instrucción.
     * Cada instrucción debe implementar este método para generar su representación en el AST.
     * @param dotGenerator - Instancia del DotGenerator, que se utiliza para crear nodos y conexiones.
     * @returns string - El identificador del nodo generado en el DOT.
     */
    public abstract generateNode(dotGenerator: DotGenerator): string;
}
```

Figura 2. Definición de la clase Instrucción.

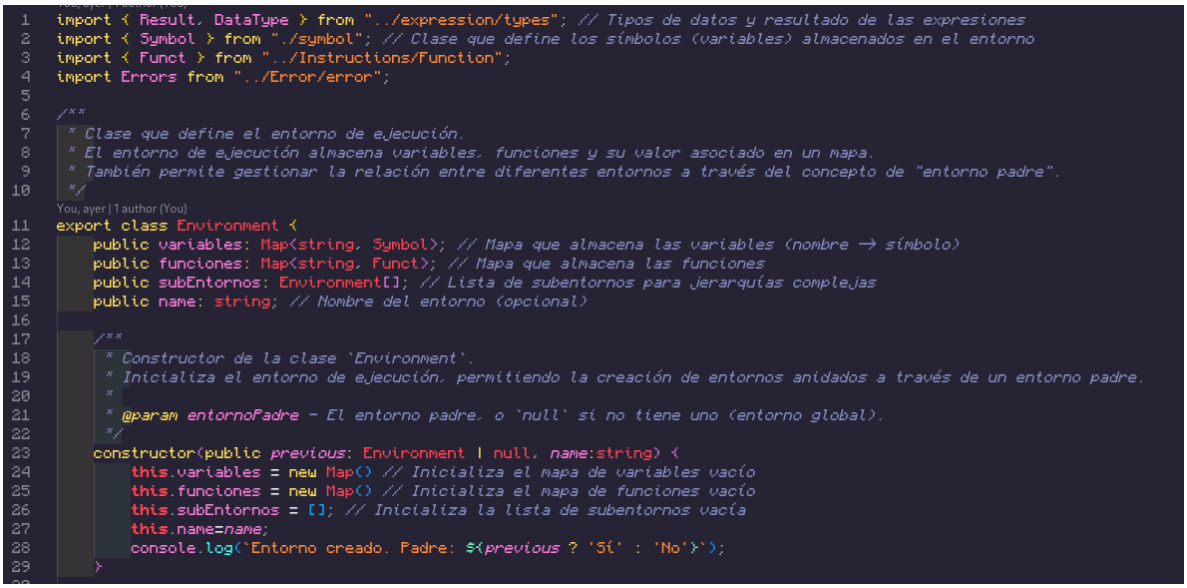
Fuente: Elaboración propia.



```
import { Environment } from "../Environment/environment";  // Entorno donde se ejecutan las expresiones
import { Result } from "../expression/types";
import { DotGenerator } from "../Tree/DotGenerator";             // Resultado de la evaluación de la expresión

/**
 * Clase abstracta que representa una expresión en el lenguaje.
 * Las expresiones son unidades de código que, cuando se evalúan, devuelven un valor.
 * Esta clase es abstracta porque define la estructura que las subclases deben seguir, pero no implementa directamente el método `execute`.
 */
You, hace 2 semanas | 1 author (You)
export abstract class Expression {
    protected linea: number;    // Número de línea donde se encuentra la expresión en el código fuente
    protected columna: number; // Número de columna donde se encuentra la expresión en el código fuente

    /**
     * Constructor de la clase `Expression`.
     * @param linea - La línea de código donde se define la expresión (para manejo de errores y depuración).
     * @param columna - La columna de código donde se define la expresión (para manejo de errores y depuración).
     */     You, hace 3 semanas • Agregar clase Expression para representar una e...
    constructor(linea: number, columna: number) {
        this.linea = linea;    // Asigna la línea donde se encuentra la expresión
        this.columna = columna; // Asigna la columna donde se encuentra la expresión
    }

    /**
     * Método abstracto que debe ser implementado por todas las clases que hereden de `Expression`.
     * Evaluar la expresión y devolver un resultado.
     *
     * @param entorno - El entorno actual donde se evaluará la expresión, permitiendo acceso a variables y funciones.
     * @returns Result - El valor resultante de la evaluación de la expresión, que incluye tanto el valor como el tipo de dato.
     */
    public abstract execute(entorno: Environment): Result;

    /**
     * Método general para generar nodos en formato DOT para Graphviz.
     * Este método puede ser reutilizado por cualquier expresión para crear su nodo.
     *
     * @param ast - Referencia al AST, que contiene el contador de nodos.
     * @param label - La etiqueta que se mostrará en el nodo.
     * @param children - Los nodos hijos conectados a este nodo (opcional).
     * @returns string - Representación en formato DOT del nodo y sus conexiones.
     */
    /**
     * Método abstracto para generar el nodo DOT de la expresión.
     * Este método debe ser implementado por cada subclase de `Expression`.
```

Figura 3. Definición de la clase Expresion.

Fuente: Elaboración propia.



```
import { Result, DataType } from "../expression/types"; // Tipos de datos y resultado de las expresiones
import { Symbol } from "./symbol"; // Clase que define los símbolos (variables) almacenados en el entorno
import { Funct } from "../Instructions/Function";
import Errors from "../Error/error";

/**
 * Clase que define el entorno de ejecución.
 * El entorno de ejecución almacena variables, funciones y su valor asociado en un mapa.
 * También permite gestionar la relación entre diferentes entornos a través del concepto de "entorno padre".
 */
You, ayer | 1 author (You)
export class Environment {
    public variables: Map<string, Symbol>; // Mapa que almacena las variables (nombre → símbolo)
    public funciones: Map<string, Funct>; // Mapa que almacena las funciones
    public subEntornos: Environment[]; // Lista de subentornos para jerarquías complejas
    public name: string; // Nombre del entorno (opcional)

    /**
     * Constructor de la clase `Environment`.
     * Inicializa el entorno de ejecución, permitiendo la creación de entornos anidados a través de un entorno padre.
     *
     * @param entornoPadre - El entorno padre, o `null` si no tiene uno (entorno global).
     */
    constructor(public previous: Environment | null, name:string) {
        this.variables = new Map() // Inicializa el mapa de variables vacío
        this.funciones = new Map() // Inicializa el mapa de funciones vacío
        this.subEntornos = []; // Inicializa la lista de subentornos vacía
        this.name=name;
        console.log(`Entorno creado. Padre: ${previous ? 'Sí' : 'No'}`);
    }
```

Figura 4. Definición de la clase Environment.

Fuente: Elaboración propia.