Universidad San Carlos de Guatemala

Facultad de Ingeniería

Organización de Lenguajes y Compiladores 2, Sección N

Ing. Edgar Rubén Sabán Raxon

Aux. Daniel Enrique Santos Godoy

Segundo Semestre 2024

Manual Técnico

Proyecto 2 – OakLand

Nombre: Carlos Manuel Lima y Lima

Registro Académico: 202201524

CUI: 3009368850101

Guatemala, 28 de octubre del 2024.

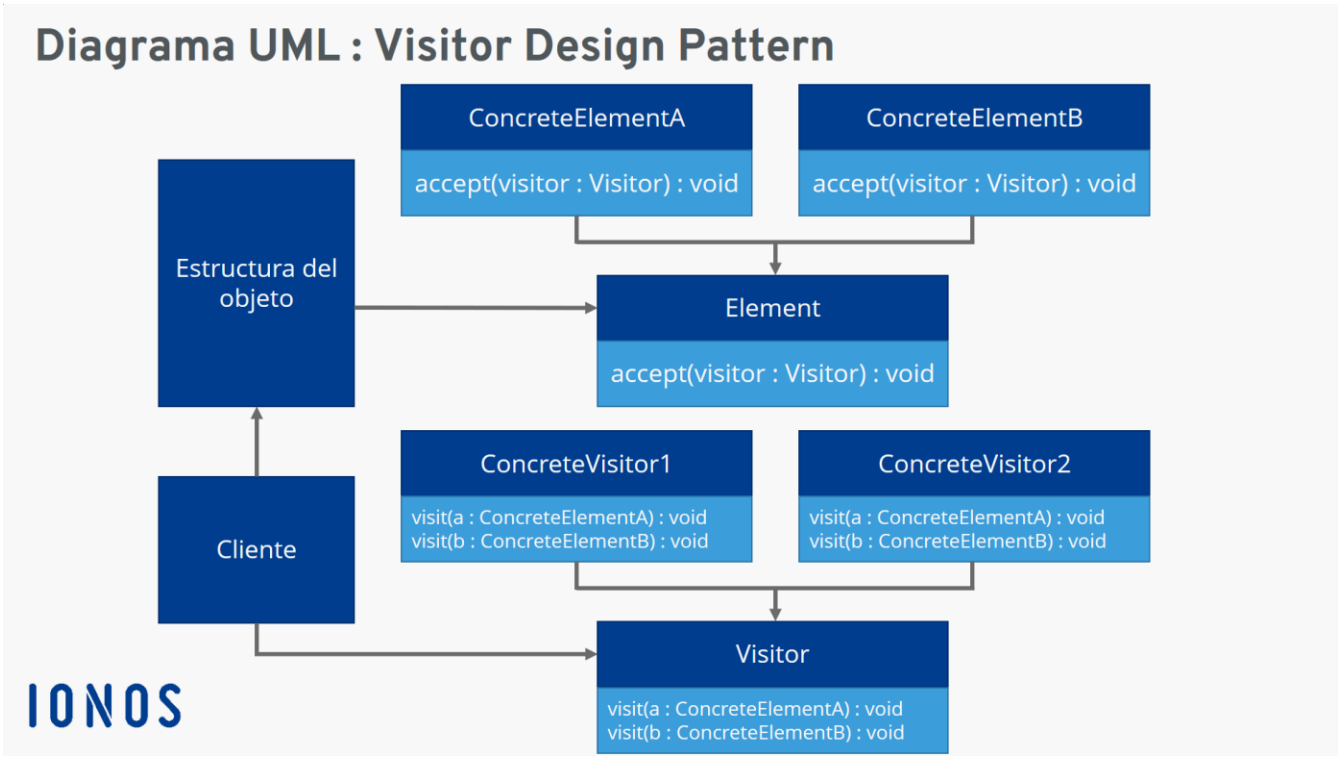
PATRÓN VISITOR

El patrón Visitor es un patrón de diseño utilizado para separar algoritmos de la estructura de datos sobre la que operan. En la creación de un intérprete, este patrón permite definir nuevas operaciones sobre una estructura de nodos sin modificar las clases de esos nodos. Es especialmente útil para recorrer árboles sintácticos abstractos generados a partir de la gramática de un lenguaje de programación.

El patrón Visitor consiste en dos elementos clave:

- Elementos visitables (nodos): Cada tipo de nodo en el AST (como expresiones, declaraciones, etc.) implementa una interfaz Visitable, que define un método para aceptar un visitante (accept(visitor)).

- El visitante (Visitor): El visitante es una clase que define un método visit para cada tipo de nodo. De esta manera, se puede agregar lógica específica para cada tipo de nodo, como evaluar expresiones o ejecutar declaraciones, sin cambiar las clases de los nodos.



CLASE VISITOR

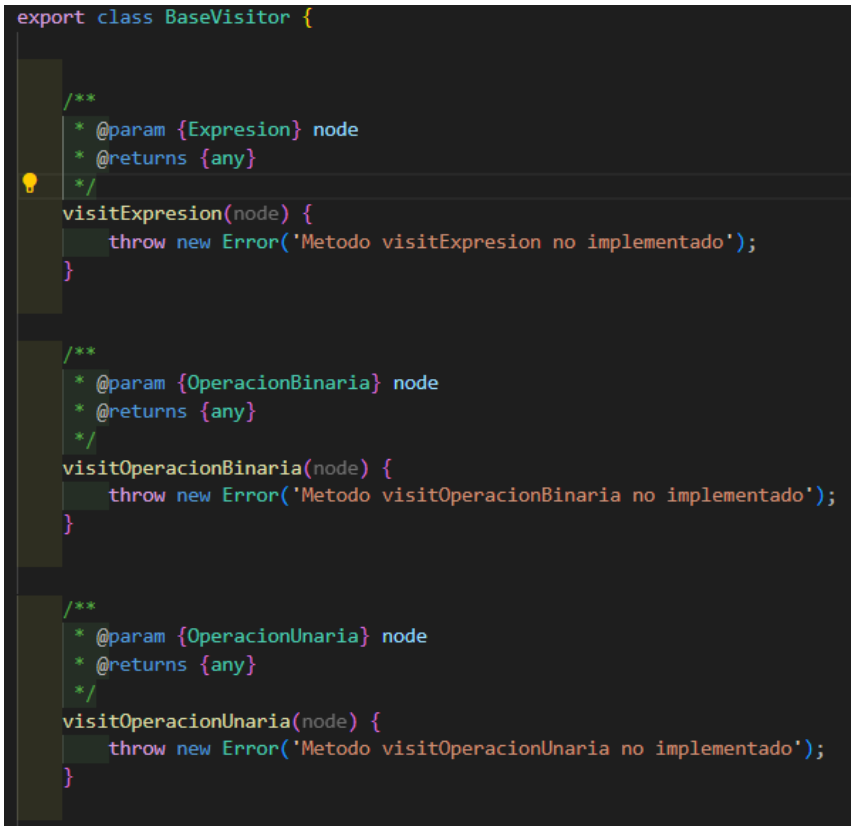
Uso de @typedef: Las anotaciones @typedef sirven para importar y declarar tipos que representan distintos nodos en el AST. Estos tipos, como OperacionBinaria, DeclaracionVar, o Print, corresponden a diferentes construcciones del lenguaje (por ejemplo, operadores binarios, declaraciones de variables, y declaraciones de impresión).

Patrón Visitor: El patrón Visitor permite definir una operación sin cambiar las clases de los elementos sobre los que opera. En este caso, cada método visitX está diseñado para procesar un tipo específico de nodo del AST, como expresiones, estructuras de control (if, while), asignaciones, o acceso a variables y arreglos.

Métodos de visita: Cada método en BaseVisitor está declarado para visitar y procesar un tipo de nodo específico, como visitOperacionBinaria para operadores binarios o visitForEach para ciclos foreach. Estos métodos son abstractos, lo que significa que se espera que una subclase los implemente para proporcionar funcionalidad específica.

Errores por defecto: Si algún método de visita no es implementado en una subclase, se lanzará un error (throw new Error) indicando que el método no está implementado. Esto asegura que todas las subclases que hereden de BaseVisitor deben proporcionar una implementación para los métodos relevantes.

Soporte para estructuras complejas: El diseño cubre una amplia gama de estructuras comunes en lenguajes de programación, como arreglos, matrices, funciones, ciclos (for, while), y más. Además, se incluyen métodos para manejar operaciones sobre arreglos (visitJoinArreglo, visitLengthArreglo) y matrices (visitAccesoMatriz, visitAsignacionMatriz), lo que refleja un soporte robusto para manipulación de datos.



```
export class BaseVisitor {


    /**
     * @param {Expresion} node
     * @returns {any}
     */
    visitExpresion(node) {
        throw new Error('Metodo visitExpresion no implementado');
    }


    /**
     * @param {OperacionBinaria} node
     * @returns {any}
     */
    visitOperacionBinaria(node) {
        throw new Error('Metodo visitOperacionBinaria no implementado');
    }


    /**
     * @param {OperacionUnaria} node
     * @returns {any}
     */
    visitOperacionUnaria(node) {
        throw new Error('Metodo visitOperacionUnaria no implementado');
    }
```

CLASE NODOS

En el archivo `nodos.js`, estás implementando una jerarquía de nodos para representar diversas estructuras sintácticas y expresiones endel lenguaje. Cada nodo corresponde a un tipo de operación o estructura en el código fuente que deseas interpretar o compilar.

La clase sigue el patrón visitor, donde cada nodo tiene un método `accept` que permite que un visitante realice operaciones sobre él, como interpretación, ejecución o validación de tipos. Cada nodo almacena información relevante como el tipo de operación (por ejemplo, asignación, llamada a función, declaración de arreglo), y algunos nodos manejan listas o grupos de expresiones (como en el caso de los bloques de código o arreglos).

Este enfoque permite organizar y manipular el árbol sintáctico de manera clara y extensible, añadiendo fácilmente nuevos tipos de nodos o operaciones sin modificar la estructura existente.

1. Nodos de Expresiones

Estos nodos representan operaciones matemáticas, lógicas, o de manipulación de valores.

- ExpresionBinaria: Representa una operación binaria, como suma, resta, multiplicación, comparación, etc. Tiene dos operandos y un operador.
- ExpresionUnaria: Representa una operación unaria, como el negativo de un valor o el complemento lógico.
- LiteralEntero, LiteralDecimal, LiteralCadena, LiteralCaracter, LiteralBooleano: Representan valores literales básicos de tipos primitivos (números enteros, decimales, cadenas de texto, caracteres y booleanos).
- Variable: Representa el acceso a una variable.
- LlamadaFuncion: Representa la llamada a una función definida en el lenguaje.

2. Nodos de Control de Flujo

Estos nodos manejan la ejecución condicional y los bucles.

- If: Representa una estructura condicional if, que ejecuta un bloque de código si una condición es verdadera.
- IfElse: Extiende If, añadiendo un bloque else que se ejecuta si la condición es falsa.
- While: Representa un bucle while que repite un bloque de código mientras una condición sea verdadera.
- DoWhile: Similar a While, pero garantiza que el bloque de código se ejecutará al menos una vez antes de verificar la condición.
- For: Representa un bucle for, con inicialización, condición y actualización de una variable de control.

3. Nodos de Estructuras de Datos

Estos nodos manejan la declaración, acceso y manipulación de arreglos.

- DeclaracionArreglo: Representa la declaración de un arreglo en el lenguaje.
- AccesoArreglo: Representa el acceso a un elemento de un arreglo mediante un índice.
- ModificacionArreglo: Representa la modificación de un valor en un arreglo mediante un índice.

4. Nodos de Declaraciones y Asignaciones

Estos nodos gestionan la declaración de variables y funciones, así como la asignación de valores.

- DeclaracionVariable: Representa la declaración de una variable con su tipo y valor inicial (opcional).
- DeclaracionFuncion: Representa la declaración de una función, con parámetros y un bloque de código asociado.
- AsignacionVariable: Representa la asignación de un valor a una variable existente.
- Return: Representa la instrucción return, que devuelve un valor desde una función.

5. Nodos de Estructuras Avanzadas

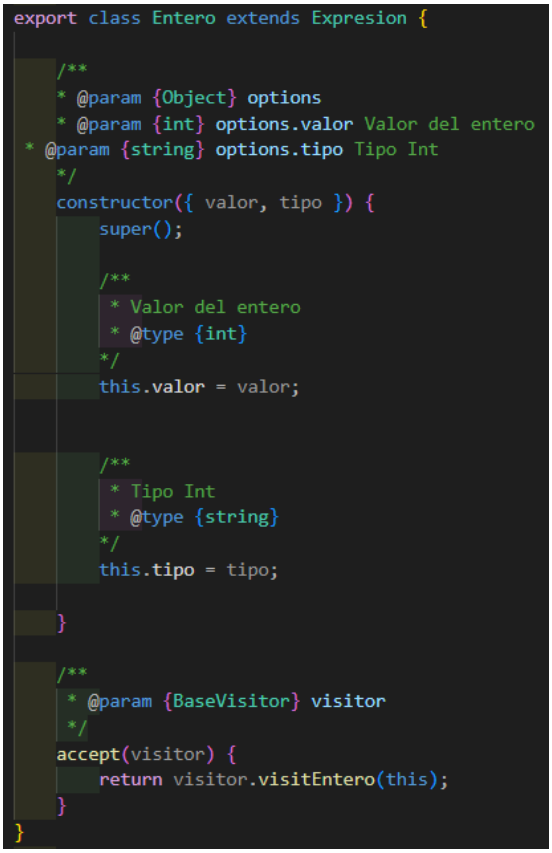
Estos nodos representan características más complejas del lenguaje.

- Struct: Representa una estructura o tipo compuesto que agrupa varios atributos bajo un solo nombre.
- AccesoAtributo: Representa el acceso a un atributo de un struct.
- AsignacionAtributo: Representa la asignación de un valor a un atributo de un struct.

6. Nodos de Sentencias de Bloque

Estos nodos agrupan sentencias y expresiones en un solo bloque de código.

- Bloque: Representa un conjunto de sentencias agrupadas en un bloque de código, como el cuerpo de un if, while, for, o una función.



```
export class Entero extends Expresion {

    /**
    * @param {Object} options
    * @param {int} options.valor Valor del entero
 * @param {string} options.tipo Tipo Int
    */
    constructor({ valor, tipo }) {
        super();

        /**
         * Valor del entero
         * @type {int}
        */
        this.valor = valor;


        /**
         * Tipo Int
         * @type {string}
        */
        this.tipo = tipo;

    }

    /**
     * @param {BaseVisitor} visitor
     */
    accept(visitor) {
        return visitor.visitEntero(this);
    }
}
```

CLASE INTERPRETE

Este intérprete utiliza el patrón **Visitor** en conjunto con un árbol sintáctico abstracto (AST) para recorrer y evaluar diferentes tipos de nodos, como operaciones, declaraciones de variables y literales. El patrón **Visitor** permite separar la lógica de ejecución de las estructuras de datos, permitiendo que el intérprete pueda manejar distintos nodos del AST sin tener que modificar sus definiciones internas.

Funcionamiento del intérprete

- **BaseVisitor**: Proporciona la interfaz que define métodos para visitar cada tipo de nodo. Los nodos del AST tienen un método accept que llama al método apropiado del visitante. Este mecanismo asegura que cada tipo de nodo sea tratado adecuadamente por el intérprete.
- **Entorno**: Es la estructura que mantiene el estado de las variables y funciones durante la ejecución del programa. Cada vez que se declara una nueva variable o se ejecuta una función, la información se almacena en el entorno actual. Si se busca una variable, el intérprete la recupera de este entorno.
- **Nodos del AST**: Cada nodo representa una construcción del lenguaje, como expresiones binarias, unarias, declaraciones de variables, literales (números, cadenas, etc.), y funciones embebidas. Los nodos tienen el rol de transportar la información sintáctica que el intérprete debe evaluar o ejecutar.

Ejecución de Operaciones

- **Operaciones Binarias y Unarias**: El intérprete evalúa las operaciones binarias y unarias visitando los nodos correspondientes y aplicando el operador especificado. Para las operaciones binarias, se visitan las subexpresiones (izquierda y derecha), y luego se ejecuta la operación combinando ambos valores. En las operaciones unarias, solo se evalúa una expresión antes de aplicar el operador.
- **Literales**: Cuando el intérprete encuentra un nodo literal (números, cadenas, booleanos), simplemente devuelve el valor de dicho nodo, indicando también su tipo. Estos nodos no requieren más procesamiento porque ya contienen un valor final.

Declaración y Referencia de Variables

- **Declaración de variables**: El intérprete maneja la declaración de variables verificando que el identificador no sea una palabra reservada y que no haya conflictos de tipos en la asignación. Si la variable cumple con las condiciones, se agrega al entorno actual. Si el tipo de la variable es var, se determina en tiempo de ejecución basado en el valor de la expresión asignada.
- **Acceso a variables**: Cuando se encuentra una referencia a una variable, el intérprete consulta el entorno actual para recuperar su valor. Si la variable no está definida, se lanza un error.

Funciones Embebidas

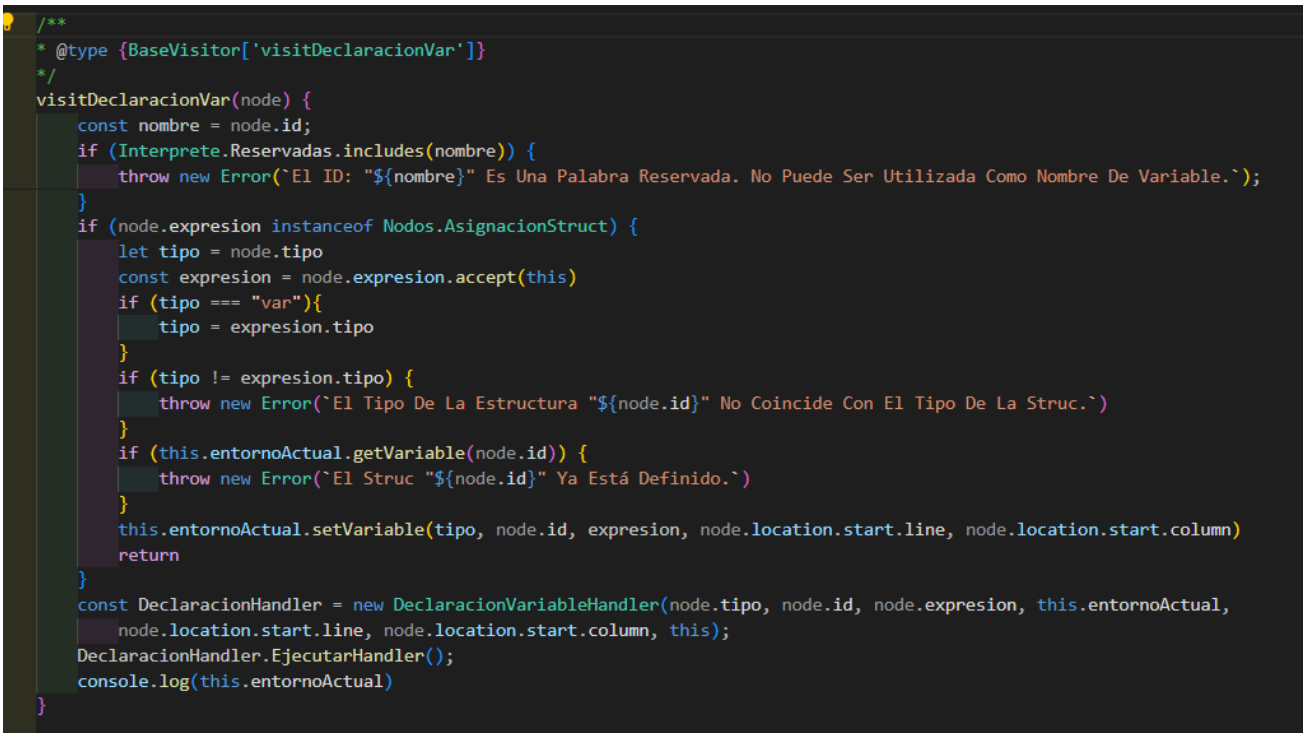
Las funciones embebidas (predefinidas) se agregan al entorno en el constructor del intérprete. Estas funciones, como toString, parseInt, entre otras, están disponibles para su uso dentro del programa y son llamadas como cualquier otra función. Al invocar una función embebida, el intérprete la busca en el entorno y ejecuta su implementación.

Evaluación de Expresiones

El intérprete evalúa las expresiones recorriendo los nodos que las representan. Cuando una expresión es una operación (binaria o unaria), el intérprete evalúa recursivamente las subexpresiones y aplica el operador correspondiente. Si es un literal, devuelve el valor inmediatamente. Si la expresión es una variable, su valor se obtiene del entorno. Las expresiones agrupadas se resuelven evaluando la subexpresión interna.

Salida

El intérprete acumula la salida del programa en una variable interna, concatenando los resultados de las llamadas a print. Esta salida se puede mostrar o procesar al finalizar la ejecución del programa.



```
/**
* @type {BaseVisitor['visitDeclaracionVar']}
*/
visitDeclaracionVar(node) {
    const nombre = node.id;
    if (Interprete.Reservadas.includes(nombre)) {
        throw new Error(`El ID: "${nombre}" Es Una Palabra Reservada. No Puede Ser Utilizada Como Nombre De Variable.`);
    }
    if (node.expresion instanceof Nodos.AsignacionStruct) {
        let tipo = node.tipo
        const expresion = node.expresion.accept(this)
        if (tipo === "var"){
            tipo = expresion.tipo
        }
        if (tipo != expresion.tipo) {
            throw new Error(`El Tipo De La Estructura "${node.id}" No Coincide Con El Tipo De La Struc.`)
        }
        if (this.entornoActual.getVariable(node.id)) {
            throw new Error(`El Struc "${node.id}" Ya Está Definido.`)
        }
        this.entornoActual.setVariable(tipo, node.id, expresion, node.location.start.line, node.location.start.column)
        return
    }
    const DeclaracionHandler = new DeclaracionVariableHandler(node.tipo, node.id, node.expresion, this.entornoActual,
        node.location.start.line, node.location.start.column, this);
    DeclaracionHandler.EjecutarHandler();
    console.log(this.entornoActual)
}
```

ENTORNO

La clase Entorno es una implementación de una tabla de símbolos para gestionar variables, estructuras y sus ámbitos.

Estructura básica:

- La clase utiliza un objeto valores para almacenar variables.
- Mantiene una referencia a un padre, permitiendo ámbitos anidados.

Gestión de variables:

- setVariable: Añade una nueva variable al entorno actual.
- getVariable: Busca una variable en el entorno actual y en los padres si es necesario.
- assignVariable: Actualiza el valor de una variable existente, con comprobación de tipos.

Manejo de tipos:

- Implementa un sistema de tipos estático con conversiones implícitas limitadas.
- Los tipos soportados incluyen: string, int, float, char, boolean.

Manejo de errores:

- Utiliza excepciones para manejar errores como redefinición de variables, asignaciones de tipo incorrecto, o acceso a variables no definidas.

Características avanzadas:

- Soporta funciones nativas y "foráneas.
- Implementa una lógica recursiva para manejar estructuras anidadas.

GRAMÁTICA

Estructura general:

- El programa comienza con PROGRAMA, que consiste en una serie de instrucciones.
- Las instrucciones se dividen en SENTENCIA y DECLARACION.

Tipos de datos:

- Soporta tipos básicos como int, float, string, boolean, char.
- Incluye estructuras de datos más complejas como arreglos y matrices.

Estructuras de control:

- Incluye IF, SWITCH, WHILE, FOR, y FOREACH.
- Soporta BREAK, CONTINUE, y RETURN.

Funciones y estructuras:

- Define FUNCIONFORRANEA para declaraciones de funciones.
- Incluye STRUCT para definir tipos de datos personalizados.

Expresiones:

- Soporta operaciones aritméticas, lógicas, relacionales, y de igualdad.
- Incluye operadores unarios y ternarios.

Características especiales:

- Soporta acceso y asignación a atributos de estructuras.
- Incluye funciones embebidas como typeof, toString, y Object.keys.
- Maneja operaciones específicas de arreglos como indexOf, join, y length.

Manejo de errores y comentarios:

- Incluye reglas para espacios en blanco y comentarios (de línea y de bloque).

Sistema de nodos:

- Utiliza una función NuevoNodo para crear nodos del árbol sintáctico abstracto (AST).

Esta gramática proporciona una base sólida para un lenguaje de programación con tipado estático, estructuras de control comunes, y algunas características avanzadas como manipulación de estructuras y arreglos.