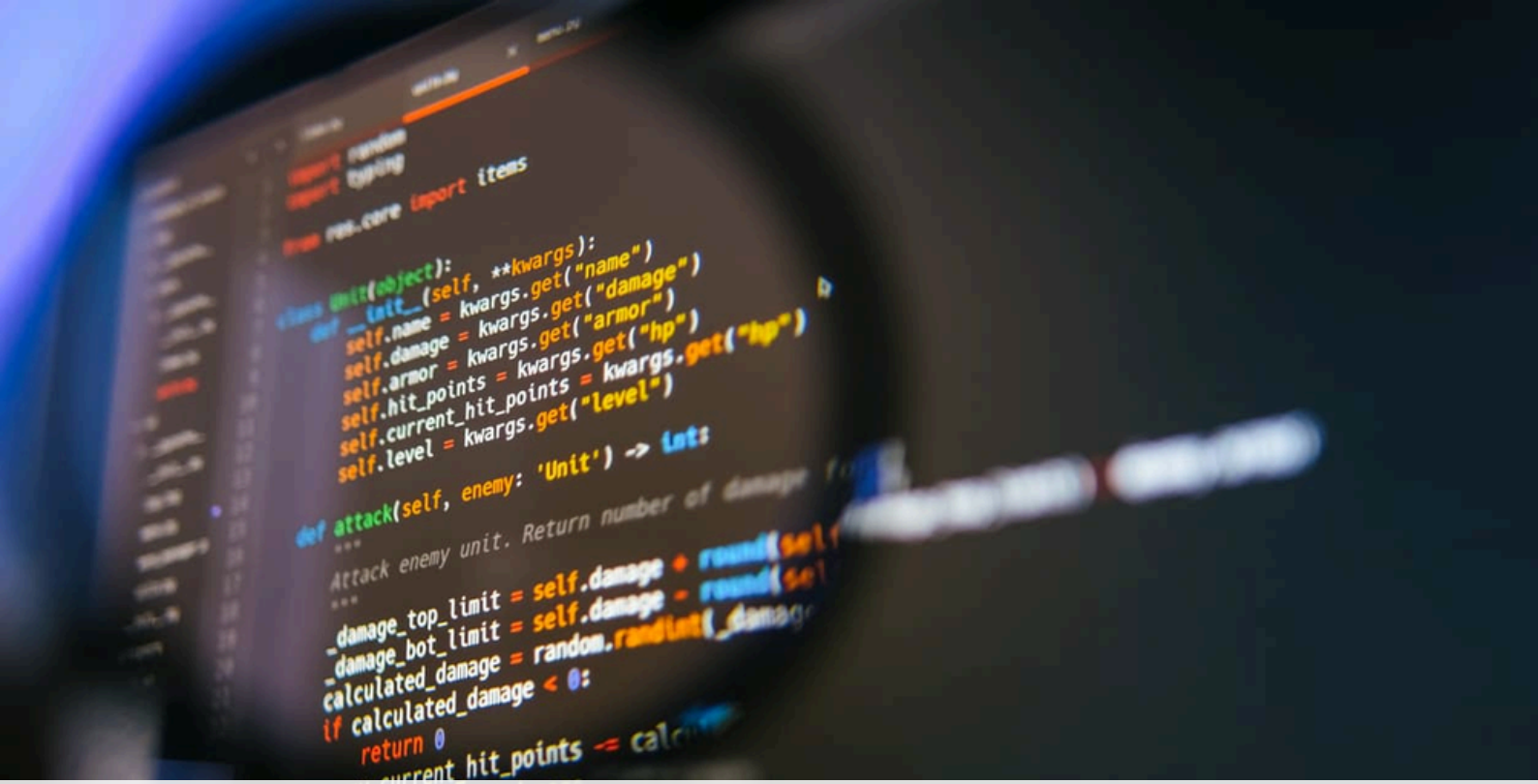
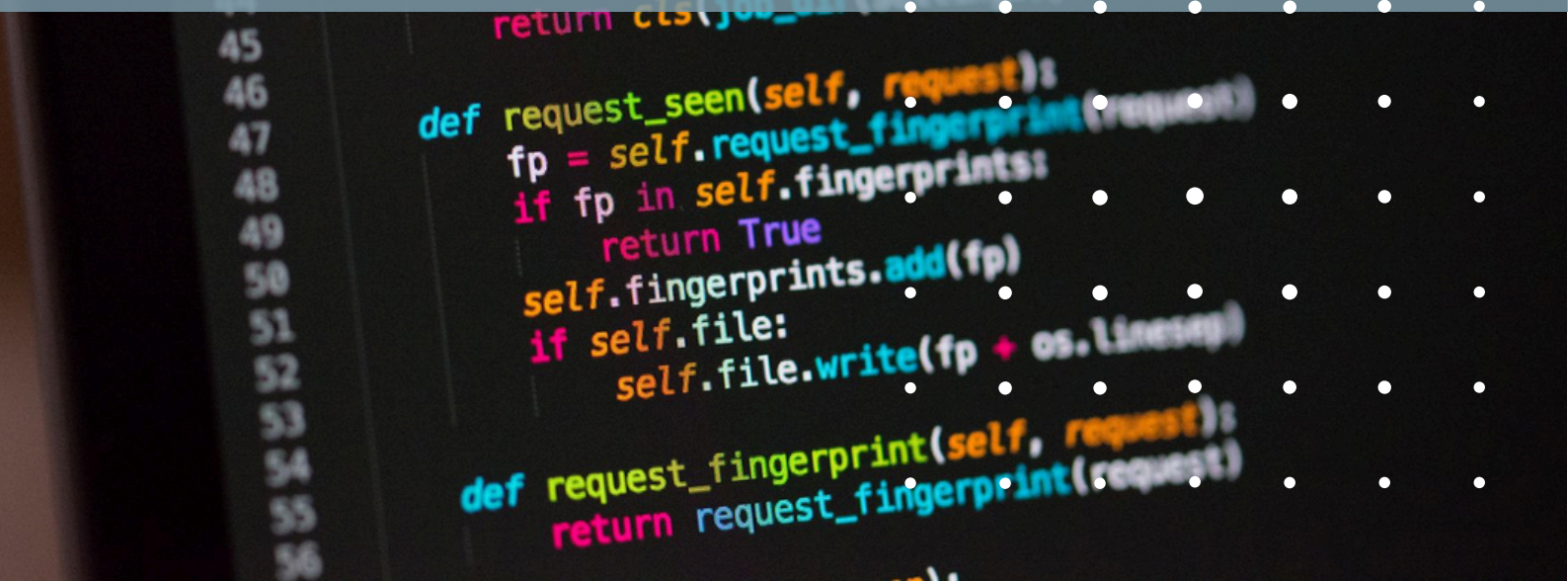


MANUAL TECNICO

ALLAN GÓMEZ

ORGANIZACION DE LENGUAJES Y COMPILADORES 1



Introducción

Este manual técnico tiene como finalidad dar a conocer al lector que pueda requerir hacer modificaciones futuras al software el desarrollo de la aplicación denominada “CompiScript+”. Para cumplir con el objetivo propuesto se incluye la descripción de las pantallas que el usuario manejara para el ingreso de datos, manejo de la simulación y de resultados, todo esto a través de gráficos para su mayor compresión. La aplicación tiene el objetivo de cumplir con los requerimientos solicitados por el curso de Organización de Lenguajes y Compiladores 1, perteneciente a la Facultada de Ingeniería de la Universidad de San Carlos de Guatemala. Dichos requerimientos incluyen el análisis de los datos de entrada con un formato previamente establecido, la generación de reportes entre otras funcionalidades.

Objetivo General

Brindar al lector una guía que contenga la información del manejo de clases, atributos, métodos y del desarrollo de la interfaz gráfica para facilitar futuras actualizaciones y futuras modificaciones realizadas por terceros.

Objetivo especifico

Mostrar al lector una descripción lo más completa y detallada posible del SO, IDE entre otros utilizados para el desarrollo de la aplicación.
Proporcionar al lector una concepción y explicación técnica - formal de los procesos y relaciones entre métodos y atributos que conforman la parte operativa de la aplicación.

Lenguaje utilizado

TypeScript es un lenguaje de programación moderno y potente que amplía y mejora JavaScript al añadirle características de tipado estático. Desarrollado y mantenido por Microsoft, TypeScript es reconocido por su capacidad para ayudar a construir aplicaciones más robustas y escalables.

Además, TypeScript se integra de forma natural con numerosas herramientas y frameworks populares de JavaScript, como Angular, React y Vue.js, lo que lo convierte en una opción poderosa para construir aplicaciones modernas y escalables en el ecosistema de desarrollo web actual.

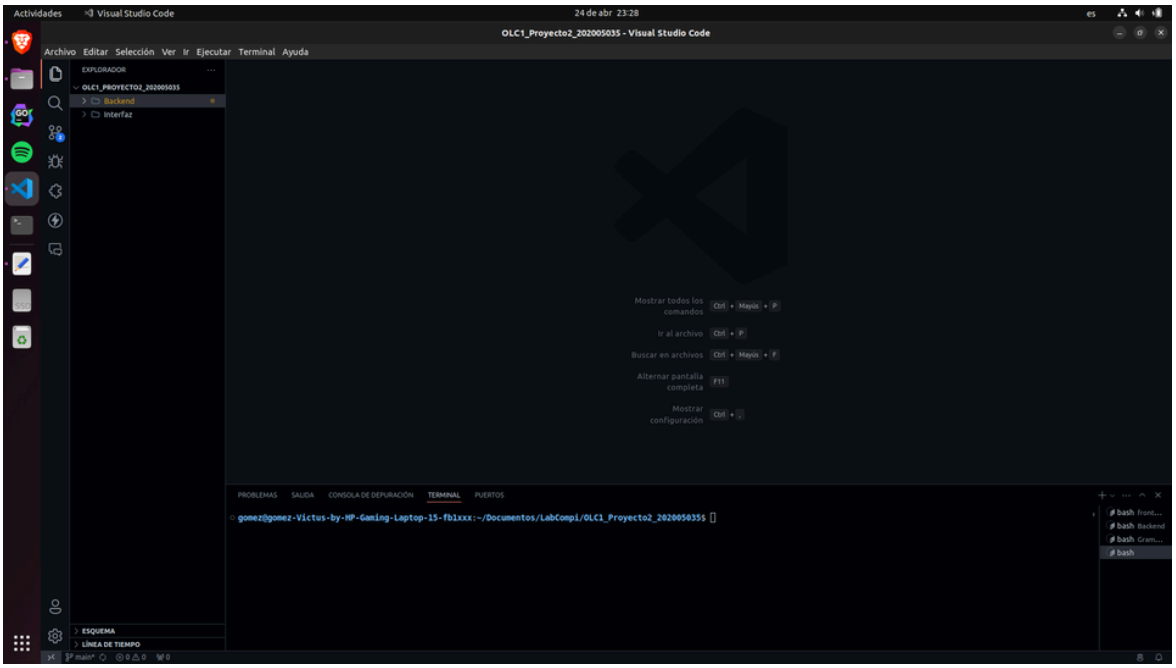
DESCRIPCIÓN DE LA SOLUCIÓN

Para poder desarrollar este proyecto se analizó lo que el cliente solicitaba y lo que el cliente realmente necesitaba, sus restricciones tanto humanas, de equipo y financieras del proyecto y empresa; y el ambiente y forma de trabajo de los futuros operadores de la aplicación. Entre las consideraciones encontramos con mayor prioridad están:

- Realizar la lectura del archivo de entrada con el formato correcto.
- Análisis completo del archivo de entrada una vez verificada que la entrada sea correcta lexicalmente, sintácticamente y semánticamente, procediendo a elaborar las respectivas funcionalidades dependiendo de las acciones que tome el usuario. Generación de reportes solicitados.
- Presentación de la interfaz de forma agradable y fácil de usar.

IDE

- El IDE con el que se desarrolló el proyecto “CompiScript+” fue Visual Studio Code, debido a su apoyo al desarrollador gracias a su asistente que detecta errores semánticos, sintácticos del código por lo cual ayudan y hacen que la duración de la fase de programación sea más corta, además posee una interfaz muy agradable y fácil de entender en el modo debugging.



Requisitos del programa

Sistema operativo	Memoria RAM mínima	Memoria RAM recomendada	Espacio en disco mínimo	Espacio en disco recomendado
El programa puede ser instalado en cualquier sistema operativo.	512 MB	8 GB	4.21 MB	1 GB

Clases del programa:

En el programa se utilizó una gran cantidad de clases, pero entre ellas las principales son:

Instrucción.ts

Es una interfaz que es la base principal de la abstracción en el proyecto, se fundamenta en que las instrucción tienen características en común y es que poseen línea y columna en donde aparecen y además todas se ejecutan pero atención que esa ejecución no posee valor de retorno.

```
import { Contexto } from "../Contexto/TablaSimbolo";

export abstract class  Instruccion{
    public line: number;
    public column: number;
    // Esta clase siempre pedirá línea y columna
    constructor(line: number, colum:number){
        this.line = line;
        this.column = colum;
    }
    // Método que siempre debe ejecutarse en todos los objetos que hereda
    public abstract interpretar(contexto:Contexto,consola:string[]):null | string

 }
```

Figura 4 Boton Ejecutar

Fuente: Elaboración Propia

Expresión.ts

Es una interfaz igual de importante que la interfaz Instrucción.ts y bastante similar con la única diferencia y es que las expresiones si retornan un valor al ejecutarse.

```ts
import { Contexto } from "../Contexto/TablaSimbolo";
import { Resultado } from "./Resultado";

export abstract class  Expresion{
    public line: number;
    public column: number;
    // Esta clase siempre pedirá línea y columna
    constructor(line: number, colum:number){
        this.line = line;
        this.column = colum;
    }
    // Método que siempre debe ejecutarse en todos los objetos que hereda
    public abstract interpretar(contexto:Contexto):Resultado

 }
```

Resultado.ts

Es una clase de tipo type que pues básicamente se utiliza para controlar los retornos así y poder verificar más fácil el valor y el tipo de los valores que retorna una expresión

```ts

export type Resultado = {
    valor: any,
    tipo: TipoDato
}

export enum TipoDato{
    NUMBER = 0,
    DOUBLE = 1,
    BOOLEANO = 2,
    CHAR = 3,
    STRING = 4,
    NULO,
    VOID
}

export enum OpAritmetica{
    SUMA, RESTA, PRODUCTO,DIVISION
}

export enum OpRelacional{
    IGUAL,DISTINTO,MENOR,MENORIGUAL,MAYOR,MAYORIGUAL
}

export enum OpLogico{
    AND,OR,NOT
}
```

TablaSimbolos.ts

Clase encargada de almacenar todas las variables que se encuentre, esta las almacena en un mapa para que sea más fácil controlar sus accesos.

```ts
import { TipoDato } from "../Expresion/Resultado";
import { Simbolo, tipoSimbolo } from "./Simbolo";

export class Contexto {
    private tablaSimbolos: Map<String,Simbolo>
    private padre: Contexto | null

    constructor(padre:Contexto | null){
        this.padre = padre;
        this.tablaSimbolos = new Map<String,Simbolo>
    }

    public guardarSimbolo(id:string,valor:Object,tipo:TipoDato,tipoSimbolo:tipoSimbolo){
        const existe = this.tablaSimbolos.has(id);
        if (!existe){
            this.tablaSimbolos.set(id, new Simbolo(id,valor,tipo,tipoSimbolo))
            console.log("Variable guardada")
            return
        }
        throw new Error("La variable ya fue declarada")
    }

    public obtenerSimbolo(id:string):Simbolo | undefined{
        let contexto_actual = this as Contexto | null
        while (contexto_actual!=null){
        const existe = contexto_actual.tablaSimbolos.has(id);
        if (existe){
            // Obtenemos valor y retornamos
            return contexto_actual.tablaSimbolos.get(id);
        }
            // Siguiente contexto
            contexto_actual = contexto_actual.padre
        }
        return undefined
    }

    public actualizarSimbolo(id:string,valor:Simbolo){
        this.tablaSimbolos.set(id,valor)
    }

    public obtenerGlobal():Contexto{
        let contexto = this as Contexto
        while(contexto.padre!=null){
            contexto = contexto.padre
        }
        return contexto
    }

    public obtenerTablaSimbolos(): Simbolo[] {
        return Array.from(this.tablaSimbolos.values());
    }
}
```

Simbolo.ts

Un símbolo es todo aquella instrucción que posea un identificador único y estos se almacenan en la TablaSimbolos

```ts
import { TipoDato } from "../Expresion/Resultado";

export class Simbolo {
    private id:string;
    private valor: Object;
    private tipo: TipoDato
    public tipoSimbolo: tipoSimbolo;

    constructor(id:string,valor:Object,tipo:TipoDato,tipoSimbolo:tipoSimbolo){
        this.id = id
        this.valor= valor
        this.tipo = tipo
        this.tipoSimbolo = tipoSimbolo
    }
    public obtenerValor():Object{
        return this.valor
    }
    public actualizarValor(valor:Object){
        this.valor = valor
    }
    public obtenertipoDato() {
        return this.tipo
    }

}

export enum tipoSimbolo {
    VARIABLE,
    FUNCION
}
```

Glosario

- **Analizador léxico:** Lee el código fuente línea por línea, carácter por carácter de izquierda a derecha y los agrupa en componentes léxicos o tokens.

- **Lenguaje:** Conjunto de cadenas o palabras de un alfabeto.

- **Lexema:** Secuencias válidas para un token que cumple el patrón.

- **Patrón:** Describe las reglas que describen la validez del token.

- **Símbolo:** Son letras, dígitos, caracteres del lenguaje.

- **Token:** Secuencia de caracteres que tienen un significado lógico colectivo.

Conclusiones

- Se debe de tener clara la idea de cómo estructurar los métodos para que estos no retornen datos incorrectos o erróneos.

- La recursividad es muy compleja por lo que se debe de tener cuidado a la hora de implementarla.

- Al analizar elemento por elemento se debe de tener claro que se está haciendo para no perderse, se debe de establecer la posición de cada elemento .

- El manejo de archivos y un amplio conocimiento sobre entorno grafico facilitan el manejo del programa.