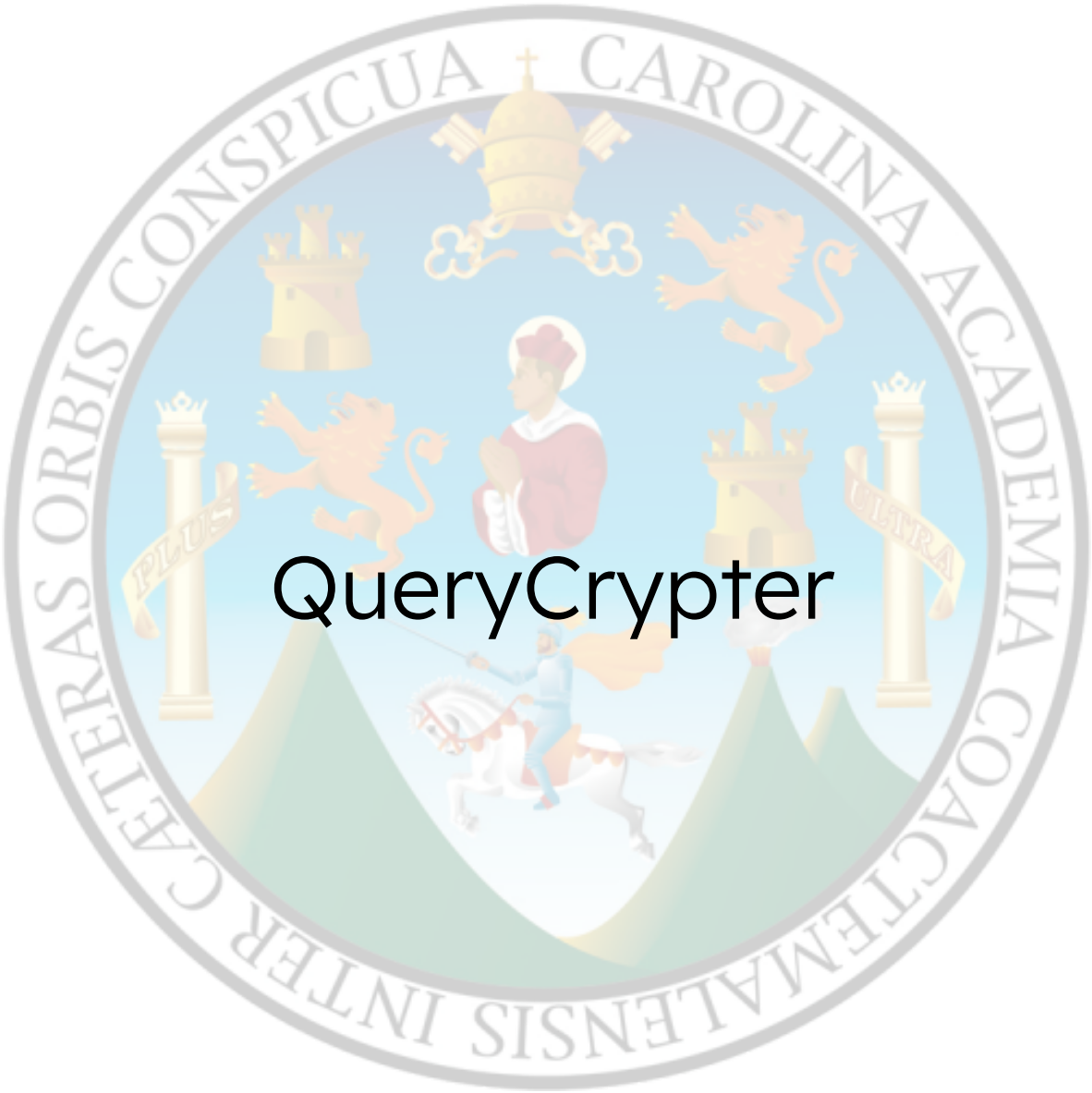
MANUAL TECNICO



QueryCrypter

Tabla de Contenido

Introducción

El presente documento describe la serie de pasos que realizan algunos de los métodos o funciones que conforman el programa computacional (QueryCrypter Interpreter) por medio de fragmentos de código, como también se detalla los conocimientos previos que debe tener el lector de este manual para comprender de una mejor manera el funcionamiento de cada una de las partes del código que conforman el programa.

Objetivos

Instruir el uso adecuado del programa computacional, describiendo el diseño y la lógica del programa por medio de fragmentos de códigos.

Describir al usuario el funcionamiento del programa para el mejor uso de él y demostrar el proceso necesario para su ejecución.

Orientar al usuario a entender la estructura del programa, como lo son sus clases y cada uno de los métodos que componen dicha clase.

Información Destacada

El manual técnico hace referencia a la información necesaria con el fin de orientar al personal en la concepción, planteamiento análisis programación e instalación del sistema. Es de notar que la redacción propia del manual técnico está orientada a personal con conocimientos en sistemas y tecnologías de información, conocimientos de programación avanzada sobre Javascript, responsables del mantenimiento e instalación del programa computacional en el computador.

Conocimientos Previos

Los conocimientos mínimos que deben tener las personas que operarán las páginas y deberán utilizar este manual son:

- Conocimientos y entendimientos en logaritmos
- Conocimientos en Javascript
- Conocimientos en HTML, CSS
- Conocimientos en Angular
- Conocimientos en NodeJs
- Conocimientos del patrón de diseño Patrón Intérprete
- Conocimiento básico de Windows

Requerimientos

El sistema puede ser instalado en cualquier sistema operativo que cumpla con los siguientes requerimientos:

- Sistema Operativo: Windows 7 o superior
- Procesador mínimo Intel Pentium (800 MHz Intel Pentium)
- Mínimo 1GB en RAM
- Visual Studio IDE, o compatible con Javascript.
- Exploradores: Internet Explorer 9 y superior

Instalación y Configuración

Para el proceso de instalación de esta aplicación únicamente es necesario tener instalado un IDE que sea compatible con el lenguaje Javascript para ejecutar la aplicación desde la terminal de este.

No es necesario tener alguna configuración ya que la configuración que trae por determinado el IDE es la necesaria para que el funcionamiento del programa sea posible.

Fragmentos de Código

En esta sección, se proporciona una explicación detallada de los métodos y funciones clave que componen el código del programa. El propósito principal es facilitar la comprensión de cómo funciona el programa, para que las personas que lo utilicen puedan brindar un mejor soporte y tener una comprensión más clara de la lógica subyacente en el software. En otras palabras, se busca documentar y explicar de manera exhaustiva las partes esenciales del código para hacer que su mantenimiento y uso sean más accesibles y comprensibles.

1. Código del método “openFile”:

```
openFile(event: any) {
  const selectedFile = event.target.files[0];
  if (selectedFile) {
    const filePath = event.target.value;
    const fileName = filePath.split('\\').pop() || filePath.split('/').pop();
    const fileReader = new FileReader();
    fileReader.onload = (e: any) => {
      const fileContent = e.target.result;
      this.tabs[this.currentTab].ruta = filePath;
      this.tabs[this.currentTab].nombre = fileName;
      this.tabs[this.currentTab].contenido_anterior = fileContent;
      this.tabs[this.currentTab].contenido_actual = fileContent;
      this.updatesLines_updateConsole();
    };
    fileReader.readAsText(selectedFile);
  }
}
```

Este método inicia su proceso verificando si el usuario ha elegido un archivo. En caso afirmativo, guarda la ubicación del archivo en la variable llamada "filePath". Si el usuario no ha seleccionado ningún archivo, la función no realiza ninguna acción adicional.

Una vez que se ha almacenado la ruta del archivo, se procede a comprobar si la extensión del archivo seleccionado es la correcta, es decir, si es ".qc". Si la extensión no es la adecuada, se mostrará un mensaje de error en la pantalla para informar al usuario del problema.

Si la extensión del archivo es correcta, el método continúa leyendo el contenido del archivo línea por línea y almacena toda esta información en una variable. Posteriormente, se mostrará el contenido completo en el área de texto de la aplicación para que el usuario pueda visualizarlo.

2. Código del método “save”:

```
async save() {
  await Swal.fire({
    title: 'Nombre Archivo',
    input: 'text',
    inputValue: this.tabs[this.currentTab].nombre + '.qc',
    inputAttributes: { spellcheck: 'false' },
    showCancelButton: true,
    confirmButtonText: 'Guardar',
    cancelButtonText: 'Cancelar',
    inputValidator: (value) => {
      if (!value) {
        return '¡Complete el campo vacio!';
      } else {
        return null;
      }
    },
  }).then((result) => {
    if (result.isConfirmed) {
      const nombreIngresado = result.value;
      this.tabs[this.currentTab].nombre = nombreIngresado;
      this.tabs[this.currentTab].contenido_anterior = this.tabs[this.currentTab].contenido_actual;
      Swal.fire('Archivo guardado exitosamente.', `Archivo guardado en C:\\Users\\Luis T\\Documents\\QueryCrypterApp\\
    }
  });
}
```

El método “save” funciona de la siguiente manera:

Este método es llamado cuando el usuario presiona el botón “Guardar” que se muestra en la interfaz gráfica de la aplicación y funciona de la siguiente manera:

Guarda todo el contenido que tiene el área de texto de la aplicación en una variable, luego verifica si el contenido pertenece a un archivo abierto por medio de la aplicación y de ser así realiza los cambios que el usuario ingresó en el archivo. En caso de que no haya un archivo abierto el programa le preguntará al usuario si desea guardar el contenido en un archivo nuevo.

3. Código del método “ejecutar”

```
run() {
  var code = this.tabs[this.currentTab].contenido_actual
  const postData = { "code": code };

  this.service.postCode(postData).subscribe(
    (response) => {
      var r = response as any;
      this.tokens = r.tokens;
      this.errors = r.errors;
      this.symbols = r.symbols;
      if (r.errors.length == 0) {
        this.linesOutputConsole = r.console.split('\n');
        this.outputConsole.nativeElement.value = r.console;
        this.outputConsoleContent = r.console;
        this.outputConsole.nativeElement.style.height = 'auto';
        this.outputConsole.nativeElement.style.height = this.outputConsole.nativeElement.scrollHeight + 'px';
      }

    },
    (error) => {
      console.error('Error:', error);
    }
  );

}
```

Este método se encarga de examinar el texto ingresado en el área de texto. En primer lugar, verifica si hay errores en el código. Si encuentra errores, se mostrará un mensaje de error en la pantalla para informar al usuario sobre la presencia de problemas.

Si no se detectan errores, el método procede a mostrar los resultados obtenidos de los comandos ingresados en la consola de entrada, en la consola de salida..

4. Código de la clase “CreateTable”:

```
export class CreateTable extends Instruction {

  constructor(line: number, column: number, private name: String, private columns: Field[]) {
    super(line, column);
  }

  public execute(environment: Environment) {
    const columns = this.columns.map((item) => {
    const value = item.execute(environment);
    return value;
    });
    environment.saveTable(this.name.toString(), new Table(this.name.toString(), columns));
  }

}
```

Esta clase es llamada al momento de crear una nueva tabla y su funcionamiento es el siguiente:

Al momento de reconocer los tokens necesarios para la creación de una tabla esta clase es llamada y se le agregan los parámetros necesarios. Al ser instanciada la clase tiene un método llamado execute el cual es llamado para que la tabla sea guardada.

5. Código de la clase “Instruction”:

```
export class CreateTable extends Instruction {

  constructor(line: number, column: number, private name: String, private columns: Field[]) {
    super(line, column);
  }

  public execute(environment: Environment) {
    const columns = this.columns.map((item) => {
    const value = item.execute(environment);
    return value;
    });
    environment.saveTable(this.name.toString(), new Table(this.name.toString(), columns));
  }

}
```

La clase que estamos describiendo tiene dos atributos llamados "line" y "column". Estos atributos se establecerán o inicializaran cuando se detecte una nueva instrucción en el analizador léxico.

6. Fragmento de código de lógica de errores:

```
for (let i = 0; i < r.errors.length; i++) {
  if (i == 0) {
    if (r.errors[i].type == "Lexico") {
      e += "Error lexico en "+r.errors[i].error+", linea "+r.errors[i].line+", columna "+r.errors[i].column+".";
    } else if (r.errors[i].type == "Sintactico") {
      e += "Error sintactico en "+r.errors[i].error+", linea "+r.errors[i].line+", columna "+r.errors[i].column+".";
    }
  } else {
    if (r.errors[i].type == "Lexico") {
      e += "\nError lexico en "+r.errors[i].error+", linea "+r.errors[i].line+", columna "+r.errors[i].column+".";
    } else if (r.errors[i].type == "Sintactico") {
      e += "\nError sintactico en "+r.errors[i].error+", linea "+r.errors[i].line+", columna "+r.errors[i].column+".";
    }
  }
}
```

Este método tiene la función de presentar los errores identificados durante el análisis léxico y sintáctico del archivo, en una ventana emergente. En otras palabras, se encarga de crear una tabla que contiene una lista detallada de los errores que se encontraron durante el proceso de análisis léxico y sintáctico del archivo. Esto permite al usuario tener una referencia clara de los problemas encontrados en el código.