



CONTENIDO

INTRODUCCION	3
DESCRIPCIÓN GENERAL DEL PROGRAMA	3
COMPETENCIAS ESPECÍFICAS ABORDADAS:	3
<i>Utilización de Herramientas para Analizadores Léxicos y Sintácticos:</i>	<i>3</i>
<i>Construcción de Intérprete para el Lenguaje X-SQL:.....</i>	<i>3</i>
<i>Traducción mediante Analizador Sintáctico Ascendente:</i>	<i>4</i>
DETALLES DE DESARROLLO.....	4
LOGICA DEL PROGRAMA.....	4
PAQUETE MANUALES:	5
PAQUETE FRONTEND	5
<i>Codemirror:</i>	<i>5</i>
<i>Img:.....</i>	<i>5</i>
<i>Editor_dinamic.js.....</i>	<i>6</i>
<i>Estilos.css</i>	<i>6</i>
<i>Index_dinamic.js</i>	<i>6</i>
PAQUETE BACKEND	9
<i>Analizadores:.....</i>	<i>9</i>
<i>BaseDatos</i>	<i>10</i>
<i>Entornos:.....</i>	<i>10</i>
<i>Expresiones:</i>	<i>14</i>
<i>Instrucciones:.....</i>	<i>15</i>
<i>Separar:.....</i>	<i>15</i>
<i>Main.py:</i>	<i>16</i>
<i>ValidacionVariasTablas.py:</i>	<i>16</i>

INTRODUCCION

Este manual, describe todos los aspectos técnicos del programa, a manera de familiarizar a la persona interesada con la lógica implementada en el desarrollo de este.

DESCRIPCIÓN GENERAL DEL PROGRAMA

El sistema XSQL es un proyecto diseñado para desarrollar un eficiente administrador de bases de datos capaz de gestionar las operaciones fundamentales de un Sistema de Gestión de Bases de Datos (DBMS) relacional convencional. El propósito principal es proporcionar a los usuarios una plataforma intuitiva que incluya un Entorno de Desarrollo Integrado (IDE) para interactuar directamente con la base de datos.

COMPETENCIAS ESPECÍFICAS ABORDADAS:

UTILIZACIÓN DE HERRAMIENTAS PARA ANALIZADORES LÉXICOS Y SINTÁCTICOS:

El programa emplea herramientas especializadas para la generación de analizadores léxicos y sintácticos, garantizando una interpretación precisa y eficiente de las instrucciones proporcionadas por el usuario.

CONSTRUCCIÓN DE INTÉRPRETE PARA EL LENGUAJE X-SQL:

El estudiante desarrolla un intérprete específico para el lenguaje X-SQL, utilizando técnicas de traducción dirigida por la sintaxis. Esto asegura la interpretación y ejecución adecuada de las consultas y comandos en el lenguaje definido.

TRADUCCIÓN MEDIANTE ANALIZADOR SINTÁCTICO ASCENDENTE:

El programa implementa un analizador sintáctico ascendente en ply para traducir y comprender las estructuras sintácticas de las consultas X-SQL, asegurando una interpretación precisa y eficiente.

DETALLES DE DESARROLLO

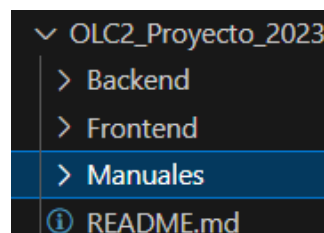
Este programa se desarrolló en lenguaje Python, a continuación, se listan las versiones del IDE y librerías utilizadas en el desarrollo:

- IDE: Visual Studio Code V 1.77.3
- JavaScript
- CSS
- HTML
- Bootstrap v.5.1.3
- d3 v.7
- Grahpviz v.3
- CodeMirror v 5.65.13.
- Python v 3.10.12
- PLY
- Flask v 3.0.0

Lo detallado anteriormente, puede asegurar el correcto funcionamiento del programa.

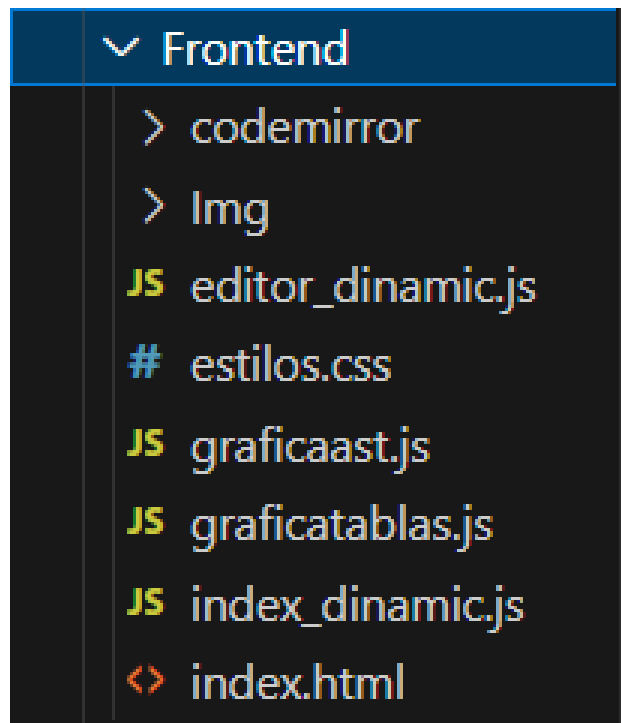
LOGICA DEL PROGRAMA

En esta sección se explicará a detalle los paquetes, clases y archivos importantes dentro del programa:



PAQUETE MANUALES:

Este paquete contine el manual de usuario, la gramática y este manual, a manera de que los usuarios o interesados puedan revisar los detalles del proyecto.

PAQUETE FRONTEND

En este paquete se puede encontrar los distintos archivos y paquetes de librerías o multimedia para la pagina web, entre estos archivos y paquetes se detallan los siguientes:

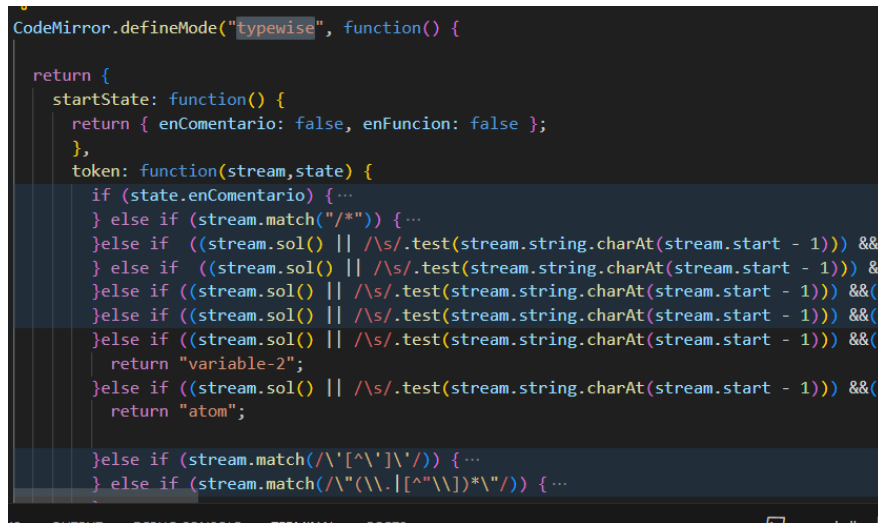
CODEMIRROR:

Este paquete o carpeta contiene todos los archivos necesarios de la librería con el mismo nombre, la cual se utiliza en la interfaz para dar estilos y aspecto similar a un editor de código a los text área correspondientes en la página web.

IMG:

Este paquete contiene todas las imágenes e iconos utilizados en la interfaz para dar mejor aspecto a la página web.

EDITOR_DINAMIC.JS



```
CodeMirror.defineMode("typewise", function() {  
  return {  
    startState: function() {  
      return { enComentario: false, enFuncion: false };  
    },  
    token: function(stream, state) {  
      if (state.enComentario) { ...  
      } else if (stream.match("/[*]") { ...  
      } else if ((stream.sol() || /\s/.test(stream.string.charAt(stream.start - 1))) &&(s  
      } else if ((stream.sol() || /\s/.test(stream.string.charAt(stream.start - 1))) &&(s  
      } else if ((stream.sol() || /\s/.test(stream.string.charAt(stream.start - 1))) &&(s  
      } else if ((stream.sol() || /\s/.test(stream.string.charAt(stream.start - 1))) &&(s  
      } else if ((stream.sol() || /\s/.test(stream.string.charAt(stream.start - 1))) &&(s  
      return "variable-2";  
      } else if ((stream.sol() || /\s/.test(stream.string.charAt(stream.start - 1))) &&(s  
      return "atom";  
    }  
    } else if (stream.match(/\\"[^\\"\\"]/) { ...  
    } else if (stream.match(/\"([^\\"\\]|\"\\\")*\\"/) { ...  
  }  
}
```

Este archivo contiene una configuración extra para el editor de texto de codemirror, prácticamente es una configuración personalizada que permite hacer un resaltado de sintaxis (colocar un color distinto a las palabras reservadas, variables, etc), además se estableció una configuración extra para que al escribir el editor de texto vaya sugiriendo la sintaxis a usar a manera de autocompletarlo al presionar enter sobre la sugerencia escogida.

ESTILOS.CSS

Este archivo contiene todos los estilos que fueron colocados a los distintos componentes de la pagina web a manera de hacer la interfaz más llamativa y responsive.

INDEX_DINAMIC.JS

Este archivo contiene el código de javascript que hace dinámica la pagina web, entre las acciones que hace están:

```
module window
var window: Window & typeof globalThis ('.navbar-nav .nav-link'); //
MDN Reference
window.addEventListener('scroll', function () {
  var logo = document.querySelector('.logoapp img');
  var navbar = document.querySelector('.navbar');

  if (window.scrollY > 30) {
    logo.src = 'Img/logoicon.ico';
    navbar.classList.add('bg-dark');
  } else {
    logo.src = 'Img/logoicon.ico';
    navbar.classList.remove('bg-dark');
  }
});
```

Este segmento hace que el navbar tenga cierto dinamismo al usar el scroll, haciendo que se seleccionen todos los elementos con la clase 'nav-link' que están dentro de elementos con la clase 'navbar-nav' y los almacena en la variable navLinks.

```
// Posicionar las secciones antes de la sección al dar clic
var links = document.querySelectorAll('.navbar-nav .nav-link');
for (var i = 0; i < 3; i++) {
  links[i].addEventListener('click', function (event) {
    event.preventDefault();
    var targetId = this.getAttribute('href').slice(1);
    var target = document.querySelector('#' + targetId);
    if (target) {
      var targetPosition = target.getBoundingClientRect().top + window.pageYOffset;
      window.scrollTo(0, targetPosition - 130);
    }
  });
}
```

Este segmento de código JavaScript se encarga de manejar eventos de clic en enlaces de navegación específicos. La funcionalidad general del código es posicionar la página para que la sección correspondiente sea visible en la parte superior del área visible del navegador cuando se hace clic en el enlace de navegación asociado.

De forma general podemos detallar la funcionalidad de lo siguiente:

1. Cambio de manuales:

- La función **changePDF** cambia el contenido del visor de PDF al archivo especificado y resalta el enlace correspondiente en la lista de manuales.

2. Mostrar offcanvas:

- Las funciones **showOffcanvas** y **showOffcanvas1** abren offcavases específicos al hacer clic en ciertos elementos.

3. Cambiar imagen de reporte:

- La función **changeImage** cambia dinámicamente la imagen en un elemento con el ID "imagenrep" basándose en el valor de texto proporcionado.

4. Funciones de Importar y Exportar:

- Las funciones **exportar** e **importar** imprimen mensajes en la consola indicando que se hizo clic en los botones de exportar e importar, respectivamente.

5. Operaciones con bases de datos:

- Las funciones **CrearBase**, **CrearDump**, y **EliminarBase** realizan operaciones relacionadas con bases de datos utilizando la API fetch para interactuar con un servidor.

6. Dinamización del menú de bases de datos:

- La función anónima dentro de **DOMContentLoaded** realiza la carga inicial del menú de bases de datos utilizando datos obtenidos del backend a través de la función **fetchDataFromBackend**. También agrega funcionalidad para expandir y contraer submenús al hacer clic.

7. Editor de consultas SQL:

- El código relacionado con el editor de consultas SQL maneja la creación dinámica de pestañas y la inicialización de editores de texto usando CodeMirror. También se definen acciones para abrir, guardar y ejecutar consultas en cada pestaña.

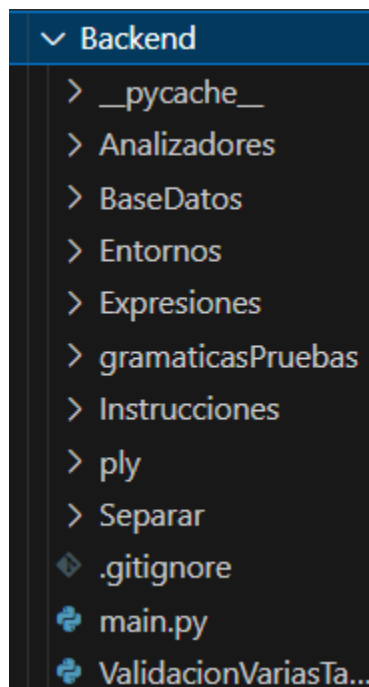
8. Funciones de manejo de pestañas:

- La función **selectTab** activa la pestaña seleccionada, y hay lógica para cerrar pestañas (excepto la primera) al hacer clic en el botón de cerrar.

9. Funciones del analizador:

- La función **ejecutaranalizador** realiza una solicitud POST al servidor con la entrada del analizador, limpia varias variables globales y realiza otras operaciones relacionadas con el análisis del código.

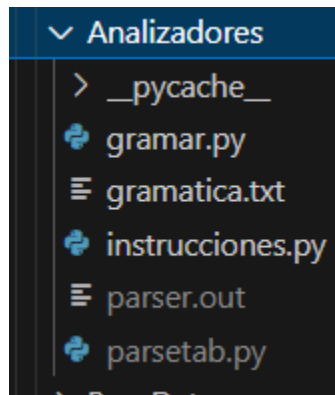
PAQUETE BACKEND



Este paquete contiene todo lo relacionado con el backend, es decir la lógica en todo el proyecto.

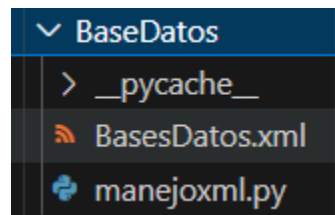
Para una mejor organización este posee otros paquetes, los cuales se detallan a continuación:

ANALIZADORES:



Aquí se encuentra todo lo relacionado al parser creado en ply, donde tenemos el archivo grammar donde se definen todos los tokens aceptados, las precedencias, expresiones regulares y producciones que ayudan a analizar una entrada, también está un archivo instrucciones que tiene la función de encapsular las distintas instrucciones analizadas para luego ser procesadas.

BASEDATOS



Esta carpeta contiene archivos relacionados con la base de datos, en otras palabras los archivos donde se almacena la información (xml) y un archivo .py en el cual se definió una clase y distintas funciones que ayudan a manejar la información del xml, es decir, se encarga de agregar información, consultar información, extraer información, eliminar o editar.

ENTORNOS:

En esta carpeta se encuentran todos los archivos que manejan los entornos, y tablas de símbolos, a continuación, se da explicación a detalle:

➤ **ENTORNO.PY:**

1. **__init__(self, padre, nombre)**: Constructor que inicializa los atributos del entorno con el entorno padre y el nombre dado.
2. **addSimbolo(self, nombre, simbolo)**: Agrega un símbolo a la tabla de símbolos.
3. **addMetodo(self, nombre, metodo)**: Agrega un método a la tabla de métodos.
4. **getSimbolo(self, nombre)**: Obtiene un símbolo buscando en el entorno actual y sus entornos padres.
5. **getSimboloE(self, nombre)**: Similar a **getSimbolo**, pero devuelve tanto el símbolo como el nombre del entorno donde se encuentra.
6. **actualizar(self, nombre, simbolo)**: Actualiza el valor de un símbolo si existe en el entorno actual o en sus padres.
7. **getMetodo(self, nombre)**: Obtiene un método buscando en el entorno actual y sus entornos padres.
8. **buscarSimboloGlobal(self, nombre)**: Busca un símbolo en todos los entornos, sin limitarse al entorno actual y sus padres.
9. **buscarSimbolo(self, nombre)**: Verifica si un símbolo está presente en el entorno actual.
10. **buscarMetodo(self, nombre)**: Verifica si un método está presente en el entorno actual.
11. **setRetorno(self, tipo)**: Establece el tipo de retorno para el entorno actual.
12. **__iter_entornos(self)**: Método privado que actúa como un generador para iterar sobre el entorno actual y sus padres.

➤ **LISTAMETODO.PY:**

1. Constructor (**__init__**):
 - Inicializa la instancia de la clase **ListaMetodo**.
 - **self.lista**: Atributo que representa la lista de objetos **RMetodo**.
2. Método **add(self, nombre, tipo)**:

- Agrega un nuevo método a la lista.
- Crea un nuevo objeto **RMetodo** con el nombre y tipo proporcionados.
- Añade el nuevo objeto **RMetodo** a la lista de métodos (**self.lista**).

➤ **LISTASIMBOLOS.PY:**

1. Constructor (**__init__**):
 - Inicializa la instancia de la clase **ListaSimbolo**.
 - **self.lista**: Atributo que representa la lista de objetos **RSimbolo**.
2. Método **add(self, nombre, contenido, tipo, entorno)**:
 - Agrega un nuevo símbolo a la lista.
 - Crea un nuevo objeto **RSimbolo** con el nombre, contenido, tipo y entorno proporcionados.
 - Añade el nuevo objeto **RSimbolo** a la lista de símbolos (**self.lista**).
3. Método **update(self, nombre, entorno, nuevo)**:
 - Actualiza el valor de un símbolo en la lista.
 - Itera sobre la lista de símbolos.
 - Si encuentra un símbolo con el mismo nombre (**id**) y entorno, actualiza su valor con el nuevo valor proporcionado (**nuevo**).

➤ **MÉTODO.PY:**

1. Constructor (**__init__**):
 - Inicializa la instancia de la clase **Metodo**.
 - **self.id**: Atributo que almacena el nombre del método.
 - **self.parametros**: Atributo que almacena la lista de parámetros del método.

- **self.instrucciones:** Atributo que almacena las instrucciones asociadas al método.
- **self.retorna:** Atributo que indica si el método retorna un valor

➤ **RMETODO.PY:**

1. Constructor (`__init__`):

- Inicializa la instancia de la clase **RMetodo**.
- **self.id:** Atributo que almacena el nombre del método.
- **self.tipo:** Atributo que almacena el tipo de retorno del método.

En resumen, la clase **RMetodo** representa un método en un programa, con información sobre su nombre y tipo de retorno. Puede ser utilizada para organizar y gestionar métodos en una estructura de datos, como en el caso de la clase **ListaMetodo** que podría contener instancias de **RMetodo**.

➤ **RSIMBOLO.PY:**

1. Constructor (`__init__`):

- Inicializa la instancia de la clase **RSimbolo**.
- **self.id:** Atributo que almacena el nombre o identificador del símbolo.
- **self.valor:** Atributo que almacena el contenido o valor asociado al símbolo.
- **self.tipo:** Atributo que almacena el tipo de dato del símbolo.
- **self.entorno:** Atributo que almacena información sobre el entorno o contexto del símbolo.

En resumen, la clase **RSimbolo** representa un símbolo en un programa con información sobre su nombre, valor, tipo y entorno. Puede ser utilizada para organizar y gestionar símbolos en una estructura de datos, como en el caso de la clase **ListaSimbolo** que podría contener instancias de **RSimbolo**.

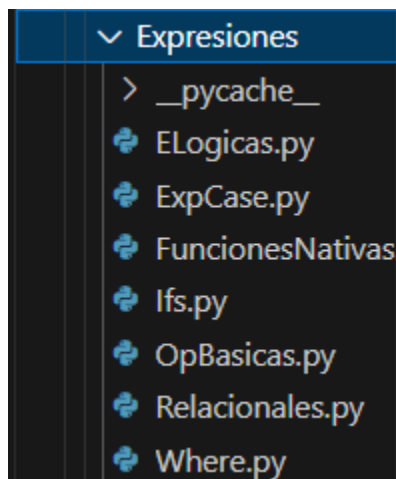
➤ **SÍMBOLO.PY:**

1. Constructor (`__init__`):

- Inicializa la instancia de la clase **Simbolo**.
- **self.id**: Atributo que almacena el nombre o identificador del símbolo.
- **self.valor**: Atributo que almacena el contenido o valor asociado al símbolo.
- **self.tipo**: Atributo que almacena el tipo de dato del símbolo.

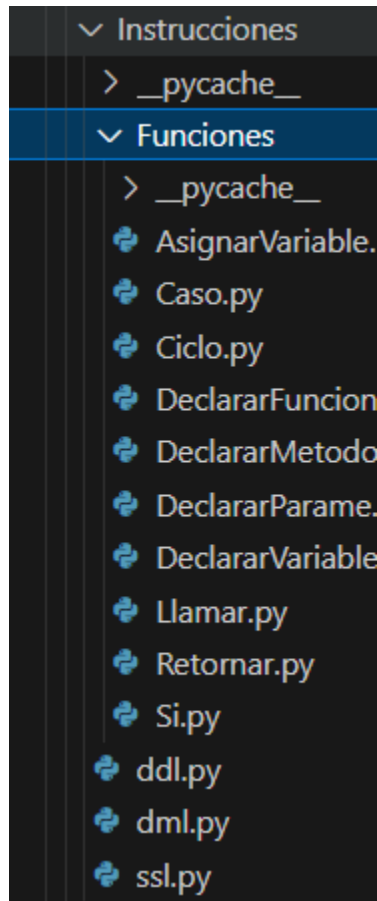
En resumen, la clase **Simbolo** representa un símbolo en un programa con información sobre su nombre, valor y tipo. Puede ser utilizada para organizar y gestionar símbolos en una estructura de datos, aunque no incluye información sobre el entorno o contexto, a diferencia de la clase **RSimbolo** que se mencionó anteriormente.

EXPRESIONES:



Aquí están todas las funciones que se encargan de procesar expresiones de la entrada, como las condicionales, expresiones lógicas, expresiones algebraicas, ifs, while, etc.

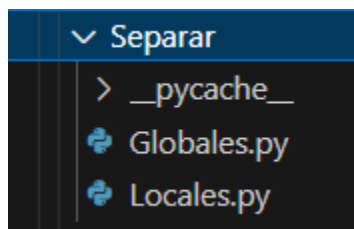
INSTRUCCIONES:



A diferencia de expresiones, este paquete maneja las instrucciones, es decir que procesa los select , créate, update, declaración de variables, ciclos, procedures, funciones, truncate, etc.

Nota: tanto de las expresiones como de las instrucciones no se dio una explicación en detalle, ya que cada función tiene una lógica propia para manejar los distintos casos, y obtener el resultado deseado.

SEPARAR:



Este paquete contiene dos archivos py, que lo que buscan es tener un control sobre en que entorno se esta trabajando las distintas instrucciones o expresiones.

En general los archivos contienen casi lo mismo, es decir una función que recibe como parámetros una lista de instrucciones, el entorno, el xml, y listas de símbolos, luego por medio de un ciclo se recorre la lista de instrucciones y dependiendo el tipo de instrucción de la que se trate se llama a alguna función de las definidas en el paquete de instrucciones o expresiones.

La diferencia entre estos radica que los locales también manejan lo relacionado con la declaración y asignación de variables.

MAIN.PY:

```
> def procesar_instrucciones(instrucciones) : ...  
  
app = Flask(__name__)  
CORS(app)  
@app.route('/a', methods=['POST'])  
✓ def procesar():  
    data = request.get_json()  
    entrada = data['entrada']  
    print(entrada)  
  
    instrucciones = g.parse(entrada.lower())  
    ts_global = TS.TablaSimbolo()  
✓    if instrucciones!=None:  
        procesar_instrucciones(instrucciones)  
        lista1=xml.recorridoarbol()  
        return jsonify({'message': 'Procesado con éxito','lista':lista1})  
  
@app.route('/crear', methods=['POST'])
```

Este archivo lo que contiene los endpoints para comunicar el backend con el frontend, a manera de enviar la información necesaria en los distintos casos.

VALIDACIONVARIAS TABLAS.PY:


```
class ComprobarTabla():
    def __init__(self):
        self.tabla = []
        self.ciclo=False
    def agregar(self, tabla):
        if self.ciclo==True:
            self.tabla.clear()
            self.ciclo=False
        self.tabla.extend(tabla)
    def comprobar(self,columnevaluar,xml,ActualBaseDatos):
        if self.tabla==None:
            print("Error: no se ha especificado una tabla")
            return -1,False,False
        igualcolumn=0
        tabref=""
        columnascomprobadas=0
        print("+++++")
        print("columna a buscar :",columnevaluar)
        for tab in self.tabla:
            if ( "." in columnevaluar)!=True: ...
            else: ...
```

Esta clase se encarga de validar los alias que puedan darse a las tablas en el select, encargándose de ver si las columnas existen en las tablas indicadas o si hay varias tablas que poseen las mismas columnas y no se a especificado por medio del alias en que tabla buscar la información.