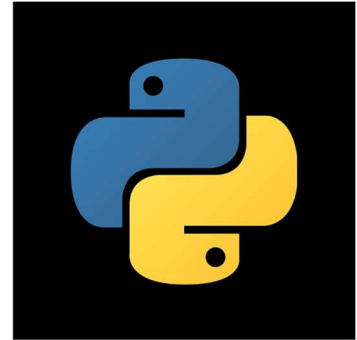


Laboratorio de Organización de Lenguajes y Compiladores 2.



PROYECTO 1 - XSQL IDE

MANUAL TECNICO

FECHA: 29/12/2023

Juan Josue Zuleta BeB

Carné: 202006353

Gerson Sebastian Quintana Berganza

Carné: 201908686

Introducción

El estudio de compiladores, en el contexto de sistemas de procesamiento de lenguaje juega un papel fundamental en la comprensión y manipulación efectiva de estructuras de lenguajes de programación. Estudiarlo, nos permite comprender cuales son las herramientas y reglas utilizadas por los compiladores e intérpretes en el procesamiento de lenguajes.

La principal razón para la realización del proyecto es poner en práctica los conocimientos teóricos adquiridos en el curso de compiladores 2. Para esto, se realizó un intérprete para un lenguaje llamado X-SQL; utilizando la herramienta para generación de analizadores léxicos y sintácticos, PLY de Python, mediante la cual se definieron las reglas del lenguaje.

A lo largo de este manual, se detallan algunas de las metodologías, herramientas utilizadas y las principales funcionalidades del sistema que proporcionan una guía detallada que no solo documenta el proceso de construcción del intérprete, sino que también sirve como recurso valioso para aquellos interesados en comprender las complejidades y desafíos asociados con el desarrollo de sistemas de procesamiento de lenguaje.

Objetivos

Instruir el uso adecuado del de la instalación y comprensión del código y de la implementación de métodos, para el acceso oportuno y adecuado en la inicialización de este, mostrando los pasos a seguir en el proceso de inicialización, así como la descripción de los archivos relevantes del sistema los cuales nos orienten en la configuración.

Requisitos Mínimos del Sistema

Sistema operativo 64 bits

- Microsoft Windows 10/8/7/Vista/2003/XP (incl.64-bit)
- macOS 10.5 o superior
- Linux GNOME o KDE desktop
- Procesador a 1.6 GHz o superior
- 1 GB (32 bits) o 2 GB (64 bits) de RAM (agregue 512 MB al host si se ejecuta en una máquina virtual)
- 3 GB de espacio disponible en el disco duro
- Disco duro de 5400 RPM
- Tarjeta de vídeo compatible con DirectX 9 con resolución de pantalla de 1024 x 768 o más.
- Navegador web (Recomendado: Google Chrome)

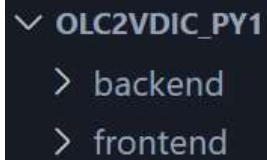
Software (Indispensable tenerlo instalado)

- IDE: Visual Studio Code
Puede conseguirse en:
<https://code.visualstudio.com/>
- Python 3.12 o posterior
Puede conseguirse en:
<https://www.python.org/downloads/>
- Node JS 21 o posterior
Puede conseguirse en:
<https://nodejs.org/en/download/current>

Descripción Técnica del Código

Estructura del proyecto

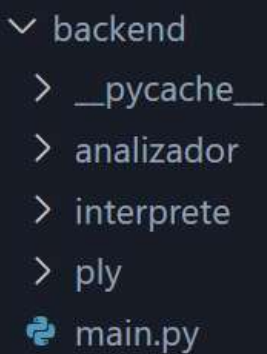
La estructura general del proyecto se compone de dos carpetas: *backend* y *frontend*.



```
▼ OLC2VDIC_PY1
  > backend
  > frontend
```

Backend

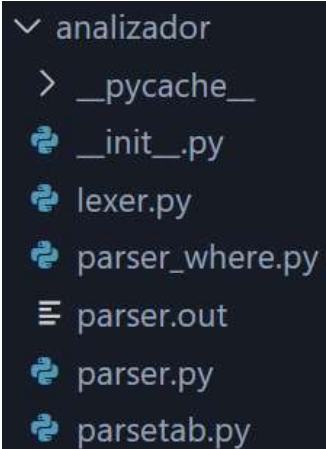
Dentro de esta carpeta se encuentran las siguientes subcarpetas:



```
▼ backend
  > __pycache__
  > analizador
  > interprete
  > ply
  main.py
```

Analizador

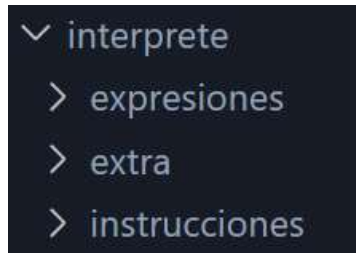
En esta carpeta se definen los analizadores léxico y sintáctico.



```
▼ analizador
  > __pycache__
  __init__.py
  lexer.py
  parser_where.py
  parser.out
  parser.py
  parsetab.py
```

Interprete

Dentro de esta carpeta se especifican las subcarpetas que serán utilizadas para funcionalidades como tal del programa.



Analizador Léxico

Para realizar el analizador léxico se utilizó la herramienta PLY, definiendo un archivo dentro de la carpeta *backend/analizador/lexer.py*. La tarea principal es dividir el código fuente en unidades más pequeñas llamadas tokens. Cada token representa una entidad léxica única, como una palabra clave, un identificador, un número o un operador.

La estructura del archivo del analizador léxico es la siguiente:

1. Definición de tokens: lista de tokens que el analizador léxico debe reconocer (incluyen palabras clave, operadores, identificadores, números, etc.).

```
reservadas = {
    'column': 'COLUMN',
    'use': 'USE',
    'create': 'CREATE',
    'data': 'DATA',
    'base': 'BASE',
    'table': 'TABLE',
    'where': 'WHERE',
    'select': 'SELECT',
    'insert': 'INSERT',
    'into': 'INTO',
    'values': 'VALUES',
    'update': 'UPDATE',
    'delete': 'DELETE',
    'drop': 'DROP',
    'truncate': 'TRUNCATE',
    'from': 'FROM',
    'as': 'AS',
    'procedure': 'PROCEDURE',
    'exec': 'EXEC',
    'function': 'FUNCTION',
    'returns': 'RETURNS',
    'return': 'RETURN',
    'begin': 'BEGIN',
    'end': 'END',
    'declare': 'DECLARE',
    'set': 'SET',
    'tokens = [
        'ENTERO',
        'CADENA',
        'FLOAT',
        'SUMAR',
        'RESTAR',
        'DIV',
        'MODULO',
        'MULT',
        'IGUAL',
        'IGUALDAD',
        'DESIGUALDAD',
        'MENOR_IGUAL',
        'MAYOR_IGUAL',
        'MENOR',
        'MAYOR',
        'OR',
        'AND',
        'NEGACION',
        'ID',
        'PARA',
        'PARC',
        'PYC',
        'PUNTO',
        'COMA',
        'ARROBA'
    ] + list(reservadas.values())
```

Como se puede ver en las imágenes anteriores, primero se especifican las palabras reservadas utilizando su *expresion_regular* : *nombre_token*, mientras que, después se especifican únicamente los nombres de los tokens, ya que estos se definirán posteriormente, y estos se unen a la tabla definida anteriormente.

2. Caracteres ignorados: cuando el analizador léxico encuentre cualquiera de los caracteres especificados en 't_ignore' los ignorará.

```
# Caracteres ignorados
t_ignore = ' \t'
```

3. Expresiones Regulares para Tokens: Como se mencionó anteriormente, hay nombres de tokens a los cuales hay que asociarle su expresión regular. Ply permite especificarlo 'dividiéndolos' en:

Expresiones regulares simples: no retornan valor del lexema.

```
# Tokens con regex
t_SUMAR = r'\+'
t_RESTAR = r'\-'
t_MULT = r'\*'
t_DIV = r'\/'
t_MODULO = r'\% '
t_PYC = r';'
t_PARA = r'\('
t_PARC = r'\)'
t_COMA = r','
t_IGUALDAD = r'=='
t_DESIGUALDAD = r'!='
t_MENOR_IGUAL = r'<='
t_MAYOR_IGUAL = r'>='
t_MENOR = r'<'
t_MAYOR = r'>'
t_IGUAL = r'='
t_OR = r'\| \| '
t_AND = r'&&'
t_NEGACION = r'!'
t_ARROBA = r'@'
t_PUNTO = r'\.'
```

Expresiones regulares complejas: retornan el valor del lexema.

```
def t_FLOAT(t):
    r'\d+\.\d+'
    try:
        t.value = t.value
    except ValueError:
        t.value = 0
    return t

def t_ENTERO(t):
    r'\d+'
    t.value = t.value
    return t

def t_ID(t):
    r'[a-zA-Z][a-zA-Z_0-9]*'
    t.type = reservadas.get(t.value.lower(), 'ID')
    return t

def t_CADENA(t):
    r'["\'][^\']*["\']'
    t.value = t.value[1:-1]
    return t
```

4. Registro del número de líneas: cuando el analizador léxico encuentra un carácter de salto de línea (`\n`), este incrementa el contador 'lineno'.

```
def t_salto_linea(t):
    r'\n+'
    t.lexer.lineno += t.value.count('\n')
```

5. Manejo de errores léxicos: cada vez que el analizador léxico encuentra un carácter no definido anteriormente, este entra a función 't_error' y lo agrega a una tabla de errores.

```
def t_error(t):
    # Agregando a la tabla de errores
    err = Error(tipo='Léxico', linea=t.lexer.lineno,
                value[t.value[0]])
    TablaErrores.addError(err)
    t.lexer.skip(1)
```

6. Creación del analizador léxico: Creación del objeto lexer utilizando `ply.lex.lex()`.

```
lex.lex()
```


Analizador Sintáctico

Para el realizar el analizador sintáctico se utilizó la herramienta PLY, definiendo un archivo dentro de la carpeta backend/analizador/parser.py. Se encarga de analizar la estructura gramatical de un programa fuente de acuerdo con las reglas definidas por la gramática del lenguaje de programación.

La estructura del archivo del analizador sintáctico es la siguiente:

1. Definición de precedencia y asociatividad de operadores.

```
precedence = (  
    ('left', 'OR'),  
    ('left', 'AND'),  
    ('left', 'MENOR', 'MAYOR', 'MENOR_IGUAL', 'MAYOR_IGUAL', 'DESIGUALDAD', 'IGUALDAD', 'IGUAL'),  
    ('left', 'RESTAR', 'SUMAR'),  
    ('left', 'MULT', 'DIV', 'MODULO'),  
    ('right', 'NEGACION'),  
    ('right', 'UMENOS')  
)
```

2. Definición de la gramática: utiliza la sintaxis de las reglas de producción de la gramática, cada regla define cómo se construye una parte del lenguaje.

```
def p_inicio(t):  
    '''  
    ini : instrucciones  
    '''  
    t[0] = t[1]  
  
def p_instrucciones(t):  
    '''  
    instrucciones : instrucciones instruccion  
    '''  
    t[1].append(t[2])  
    t[0] = t[1]  
  
def p_instrucciones_instruccion(t):  
    '''  
    instrucciones : instruccion  
    '''  
    t[0] = [t[1]]
```

```
def p_instruccion(t):
    ...
    instruccion : clausula PYC
                | declaracion_variable PYC
                | asignacion_variable PYC
                | crear_procedure PYC
                | ejecutar_procedure PYC
                | crear_funcion PYC
                | expresion PYC
                | sentencia_control PYC
    ...
    t[1].text_val += ';'
    t[0] = t[1]
```

Como se puede ver, algunas producciones cuentan con un atributo llamado 'text_val', esto, como se verá posteriormente, sirve para guardar la cadena de texto asociada a cada instrucción, esto principalmente ayuda al manejo de funciones y procedimientos almacenados.

```
def p_asignacion_variable(t):
    ...
    asignacion_variable : SET ARROBA ID IGUAL expresion
    ...
    text_val = f'SET @ {t[3]} = {t[5].text_val}'
    t[0] = AsignacionVar(text_val, t[3], t[5], t.lineno(1), t.lexpos(1))
```

Por ejemplo, en la imagen anterior se puede ver la definición sintáctica para una asignacion de una variable. El analizador, al encontrar esta estructura, ingresa a la función 'p_asignacion_variable', crea el atributo asociado a la producción 'text_val' (que es la cadena de texto asociada a la producción) y luego crea un objeto de la clase 'AsignacionVar', el cual recibe el 'text_val', el identificador de la variable, el valor de la expresión, la línea y la columna. Y esto es igual todas las reglas sintácticas.

3. Manejo de errores sintácticos: cada vez que el parser encuentra un error sintáctico, ingresa a la función 'p_error'. Cuando el valor del

token incorrecto 't' no es None, agrega el error a la tabla de errores, descarta el token y continua con el análisis sintáctico. Pero cuando el valor del token es None, significa que el parser no puede recuperarse del mismo, por lo que simplemente lo agrega a la tabla de errores y termina con el análisis sintáctico.

```
# Error sintáctico
def p_error(t):
    if t is not None:
        # Agregando a la tabla de errores
        err = Error(tipo='Sintáctico', linea=t.lineno, columna=find_column(t.lexer.lexdata, t), descripcion=f'No se esperaba token: {t.value}')
        # Se descarta el token, y el analizador continua
        parser.errok()
    else:
        # Agregando a la tabla de errores
        err = Error(tipo='Sintáctico', linea=0, columna=0, descripcion=f'Final inesperado.')
        TablaErrores.addError(err)
```

Interprete

El intérprete es una representación en clases de cada una de las instrucciones y expresiones válidas en el lenguaje X-SQL, como lo son declaraciones y asignaciones de variables, operaciones aritméticas y lógicas, etc. Estas clases se dividen en dos tipos: expresiones e instrucciones.

Expresiones

Una expresión será toda clase que al 'ejecutarse' retorna un valor. Todas estas expresiones van a heredar de otra clase llamada 'Expresion', por lo que, todas las expresiones que hereden de esta tendrán los siguientes atributos y métodos:

```
class Expresion:
    def __init__(self, text_val:str='', linea=0, columna=0) -> None:
        self.text_val = text_val
        self.linea:int = linea
        self.columna:int = columna

    def ejecutar(self, env:Enviroment):
        pass

    def ejecutar3d(self, env:Enviroment, generador:Generador):
        pass
```

El método 'ejecutar', realiza la operación correspondiente a su expresión; y el método 'ejecutar3d', traduce la expresión a su correspondiente código de 3 direcciones (para cuando aplique).

Un ejemplo para esto, es la clase 'Logica', la cual retorna el resultado de una operación lógica. Esta, tal y como se mencionó, hereda de la clase Expresion y cuenta con dos operandos 'op1' y 'op2', así como el operador. Las otras dos etiquetas únicamente sirven para generar el código de 3 direcciones de la expresión.

```
class Logica(Expression):
    def __init__(self, text_val:str, op1:Expression, operador:TipoLogico, op2:Expression, linea, columna):
        super().__init__(text_val, linea, columna)
        self.op1 = op1
        self.op2 = op2
        self.operador = operador
        self.etq_true = ''
        self.etq_false = ''
```

Asimismo, dentro del método 'ejecutar', se realizan las validaciones correspondientes, como validar que tanto el operando 1 y 2 sean de tipo booleano, así como el retorno de la expresión.

```
def ejecutar(self, env:Enviroment):
    op1:Retorno = self.op1.ejecutar(env)
    op2:Retorno = self.op2.ejecutar(env)
    resultado = Retorno(tipo=TipoDato.ERROR, valor=None)

    # Que no haya error en los operandos
    if op1.tipo == TipoDato.ERROR or op2.tipo == TipoDato.ERROR:
        # Agregando a la tabla de errores
        err = Error(tipo='Semántico', linea=self.linea, columna=self.columna, descripcion=f'Error al realizar la operacion logica.')
        TablaErrores.addError(err)
        return resultado

    if op1.tipo != TipoDato.BOOL or op2.tipo != TipoDato.BOOL:
        # Agregando a la tabla de errores
        err = Error(tipo='Semántico', linea=self.linea, columna=self.columna, descripcion=f'Ambas expresiones deben ser de tipo logicas (true o false).')
        TablaErrores.addError(err)
        return resultado

    if self.operador == TipoLogico.AND:
        resultado.tipo = TipoDato.BOOL
        resultado.valor = op1.valor and op2.valor

    elif self.operador == TipoLogico.OR:
        resultado.tipo = TipoDato.BOOL
        resultado.valor = op1.valor or op2.valor
```

Instrucciones

Una instrucción será toda clase que al 'ejecutarse' NO retorna un valor. Todas estas instrucciones van a heredar de otra clase llamada 'Instrucción', por lo que, todas las instrucciones que hereden de esta tendrán los siguiente atributos y métodos:

```
class Instruccion:
    def __init__(self, text_val:str='', linea: int = 0, columna: int = 0) -> None:
        self.text_val = text_val
        self.linea: int = linea;
        self.columna: int = columna;

    def ejecutar(self, env:Enviroment):
        pass

    def ejecutar3d(self, env:Enviroment, generador:Generador):
        pass
```

El método 'ejecutar', realiza la operación correspondiente a su instrucción; y el método 'ejecutar3d', traduce la instrucción a su correspondiente código de 3 direcciones (para cuando aplique).

Un ejemplo para esto, es la clase 'Print', la cual simplemente realiza una instrucción (determinada por el lenguaje). Esta, tal y como se mencionó, hereda de la clase Instrucción y únicamente cuenta con los atributos heredados. Las otras dos etiquetas únicamente sirven para generar el código de 3 direcciones de la expresión.

```
class Print(Instruccion):
    def __init__(self, text_val:str, argumento, linea, columna):
        super().__init__(text_val, linea, columna)
        self.text_val = text_val
        self.argumento = argumento
```

Asimismo, dentro del método 'ejecutar', se realizan las validaciones correspondientes, como validar que el argumento no sea de un tipo definido como 'ERROR' o agregar el valor de la expresión a una consola.

```
def ejecutar(self, env:Enviroment):
    exp = self.argumento.ejecutar(env)

    # Validar que no haya un error en la expresion
    if exp.tipo == TipoDato.ERROR:
        print("Semántico", f'Error en la expresión de la funcion print()', self.linea, self.columna)
        return self

    Consola.addConsola(exp.valor)
```

AST

Para la generación del AST (Árbol Sintáctico Abstracto), se construyó un árbol n-ario, y es decir, cada nodo el árbol puede tener 'n' hijos, y cada nodo tiene como atributos un identificador (para las declaraciones y conexiones en graphviz), un valor (que es el texto como tal que irá dentro del nodo) y un arreglo (que será para los hijos de ese nodo); así como un conjunto de métodos asociados.

```
class Nodo:
    # id -> Identificador del nodo
    # valor -> Contenido del nodo
    # hijos -> Arreglo de 'n' hijos
    def __init__(self, id, valor:str, hijos):
        self.id = id
        self.valor = valor
        self.hijos = hijos

    def addHijo(self, nodoHijo):
        self.hijos.append(nodoHijo)

    def getId(self):
        return self.id

    def getValor(self):
        return self.valor

    def getHijos(self):
        return self.hijos
```

La clase 'AST' se encarga de construir la estructura del árbol, así recorrerlo para crear un archivo .dot con la estructura correcta:

```

class AST:
    id:int = 0
    def __init__(self, instrucciones):
        self.instrucciones = instrucciones

    def getAST(self):
        declaraciones = ''
        conexiones = ''
        raiz = Nodo(id=0, valor='INSTRUCCIONES', hijos=[])

        try:
            for instruccion in self.instrucciones:
                instruccion.recorrerArbol(raiz)
        except TypeError as e:
            return

        declaraciones = f'\t{raiz.getId()} [label = "{raiz.getValor()}"];\n'
        declaraciones, conexiones = self.graficarArbol(raiz, declaraciones, conexiones)
        dot = 'digraph {\n' + declaraciones + conexiones + '}\n'

```

Código de Tres Direcciones

El código de 3 direcciones únicamente se realizó para las instrucciones de: Declaración, Asignación, If-else y Select (impresión). Cada una de estas clases cuenta con un método llamado “ejecutar3d”, el cual, utiliza terminales, etiquetas, la pila y heap, para generar un código equivalente al de la entrada.

```

def ejecutar3d(self, env:Enviroment, generador:Generador):
    codigo = ''
    temp = ''

    temp = generador.obtenerTemporal()

    if self.tipo == TipoDato.DECIMAL or self.tipo == TipoDato.INT:
        codigo += f'{temp} = {self.valor};\n'

    elif self.tipo == TipoDato.BIT:
        if self.valor == True or self.valor == False:
            codigo += f'{temp} = {int(self.valor)};\n'
        else:
            codigo += f'{temp} = {0};\n'

    elif self.tipo == TipoDato.NVARCHAR or self.tipo == TipoDato.NCHAR or self.tipo == TipoDato.DATE or self.tipo == TipoDato.DATETIME:
        codigo += f'{temp} = HP;\n'
        for c in self.valor:
            codigo += f'heap[HP] = {ord(c)};\n'
            codigo += f'HP = HP + 1;\n'
        codigo += f'heap[HP] = 0;\n'
        codigo += f'HP = HP + 1;\n'

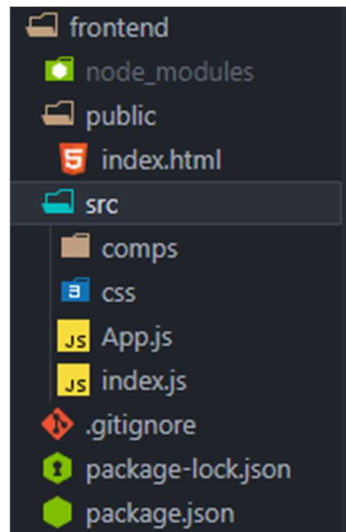
    generador.agregarInstruccion(codigo)

    return Retorno3d(codigo=codigo, etiqueta='', temporal=temp, tipo=self.tipo, valor=self.valor)

```

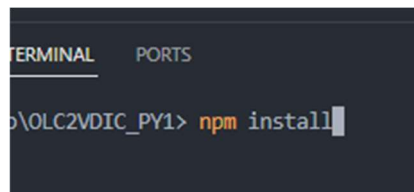

Frontend

Dentro de esta carpeta se encuentran las siguientes subcarpetas:



Node Modules

En esta carpeta se almacenan todas las dependencias del modulo de node, dichas dependencias se instalan de forma automática una vez descargado el proyecto con el siguiente comando:



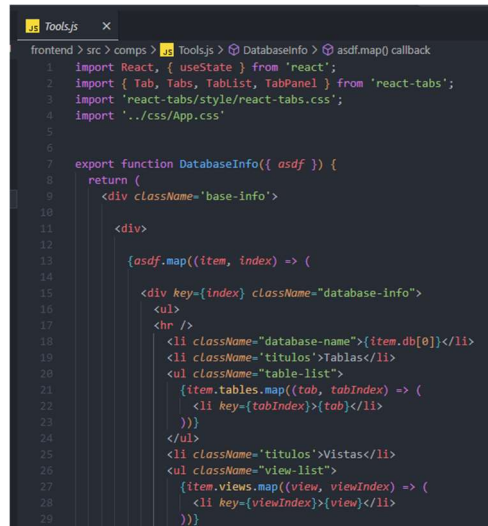
App.js

Este programa está escrito bajo el modulo de React.js el cual implementa Javascript para el funcionamiento de los componentes, dichos componentes fueron escrito en un archivo llamado Tools.js el cual se encuentra dentro de la carpeta Comps.

Cada uno de los componentes se exportan como funciones para ser renderizados dentro de la aplicación, la estructura central es basada en HTML y se despliega mediante esta sintaxis, también se implementaron estilos de hojas en cascadas y Bootstrap para la mejora del aspecto visual.

Cada uno de los componentes utilizados se describen a continuación:

Los componentes almacenados en el archivo Tools.js se almacenan como funciones exportadas que pueden ser renderizadas en cualquier archivo siempre que se importen al mismo:



```
frontend > src > comps > Tools.js > DatabaseInfo > asdf.map() callback
1 import React, { useState } from 'react';
2 import { Tab, Tabs, TabList, TabPanel } from 'react-tabs';
3 import 'react-tabs/style/react-tabs.css';
4 import '../css/App.css'
5
6
7 export function DatabaseInfo({ asdf }) {
8   return (
9     <div className='base-info'>
10
11       <div>
12
13         {asdf.map((item, index) => (
14
15           <div key={index} className='database-info'>
16             <ul>
17               <hr />
18               <li className='database-name'>{item.db[0]}</li>
19               <li className='titulos'>Tablas</li>
20               <ul className='table-list'>
21                 {item.tables.map((tab, tabIndex) => (
22                   <li key={tabIndex}>{tab}</li>
23                 ))}
24               </ul>
25               <li className='titulos'>Vistas</li>
26               <ul className='view-list'>
27                 {item.views.map((view, viewIndex) => (
28                   <li key={viewIndex}>{view}</li>
29                 ))}
30             </ul>
10
```

El estilo de hojas en cascada se almacena en la carpeta CSS la cual contienen el archivo app.css dentro del mismo se encuentran señalizados cada uno de los componentes con comentarios de forma en que pueda ser sencillo localizar cada una de las partes que se deseen revisar o modificar, dichas configuraciones corresponden a un estilo css tradicional.



```
frontend > src > css > App.css > .side_header
1 /*
2 General
3 */
4
5 * {
6   margin: 0;
7   padding: 0;
8 }
9
10 body {
11   align-items: center;
12   justify-content: center;
13   display: flex;
14   background-color: #27374D;
15 }
16
17
18 /*
19 General
20 */
21
```

La aplicación se maneja enteramente en el archivo app.js el cual contiene todos los import de las partes visuales y no estructurales de la aplicación.

```
import React, { useState } from 'react';
import './css/App.css';
import { Pestaña, DatabaseInfo } from './comps/Tools';
import { okaidia } from '@uiw/codemirror-theme-okaidia';
import CodeMirror from '@uiw/react-codemirror';
import { javascript } from '@codemirror/lang-javascript';
```

La parte central de la aplicación se basa en el return de la estructura en formato HTML hacia el index.js el cual encapsula toda la aplicación.

```
return (
  <div className="App">
    <div>
      <div className="navbar" >
        <h1 className="custom-color">XSQL-IDE</h1>
        <h1 className="custom-color">XSQL-IDE</h1>
        <h1 class="custom-color2">XSQL-IDE</h1>
        <h1 className="custom-color">XSQL-IDE</h1>

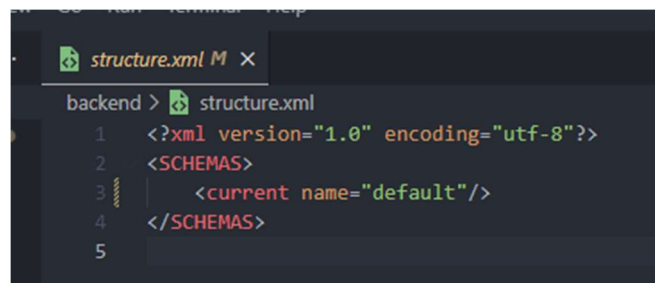
        <div className="file-input-wrapper" style={{ display: 'flex', justifyContent: 'space-b
          <input className="file-input" type="file" id="fileInput" onChange={import_data} />
          <label className="file-input-label" htmlFor="fileInput"> ABRIR </label>
          <button className="buttons" onClick={handleSave}> GUARDAR </button>
        </div>
      </div>
    </div>
  </div>
)
```

Para la implementación de la conexión con el backend se utilizo Flask y por medio de peticiones HTTP se hicieron las respectivas solicitudes a la API.

```
//barra Lateral de la base de datos
const xml_datos =
  async () => {
    fetch('http://localhost:5000/xml', {
      method: 'GET',
      headers: {
        'Content-Type': 'application/json'
      }
    })
    .then(res => res.json())
    .then(data => {
      set_current_item(data)
      setEstado(true)
      // console.log(data)
    })
    .catch(err => console.log(err))
  }
}
```

La mayor parte de la lógica de la aplicación Frontend funciona por medio del archivo llamado, structure.xml almacenado en la carpeta Backend, este archivo contiene toda la estructura de las bases de datos por lo cual cada consulta y cada uno de los componentes dependen enteramente de este, el archivo structure.xml se encarga del manejo de todos los datos como una meta data la cual se puede consultar por medio de comandos que reconoce el lenguaje, estos comando serán mostrados en forma de tablas al realizar consultas o como respuestas propias del lenguaje.

La estructura del DBMS tiene el siguiente aspecto al estar por defecto:



```
structure.xml M X
backend > structure.xml
1  <?xml version="1.0" encoding="utf-8"?>
2  <SCHEMAS>
3      <current name="default"/>
4  </SCHEMAS>
5
```

La variable *current*: almacena el nombre de la base de datos seleccionada.

La estructura cambia al ser ejecutados los comandos para crear bases de datos, por lo que una base de datos vacía tendría la siguiente estructura:



```
structure.xml M X
backend > structure.xml
1  <?xml version="1.0" encoding="utf-8"?>
2  <SCHEMAS>
3      <current name="default"/>
4      <database name="personas">
5          <tables/>
6          <views/>
7          <functions/>
8          <procedures/>
9      </database>
10 </SCHEMAS>
11
```

Sobre cada uno de los hijos de la rama database se almacenarán las tablas, vistas, funciones y procedimientos como corresponda, dicho funcionamiento y ha sido detallado anteriormente en el funcionamiento del parser.

Enlace a repositorio:

https://github.com/why-ego/OLC2VDIC_PY1