Laboratorio de Organización de Lenguajes y Compiladores 1, Sección C.

PROYECTO 2 MFMSCRIPT

MANUAL TECNICO

FECHA: 04/11/2022

Juan Josue Zuleta Beb
Carné: 202006353

Introducción

El presente documento describe los aspectos técnicos informáticos de la aplicación, deseñada a través de la interfaz gráfica de Vue.js; El documento familiariza al personal técnico especializado encargado de las actividades de mantenimiento, revisión, solución de problemas, instalación y configuración del sistema.

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.

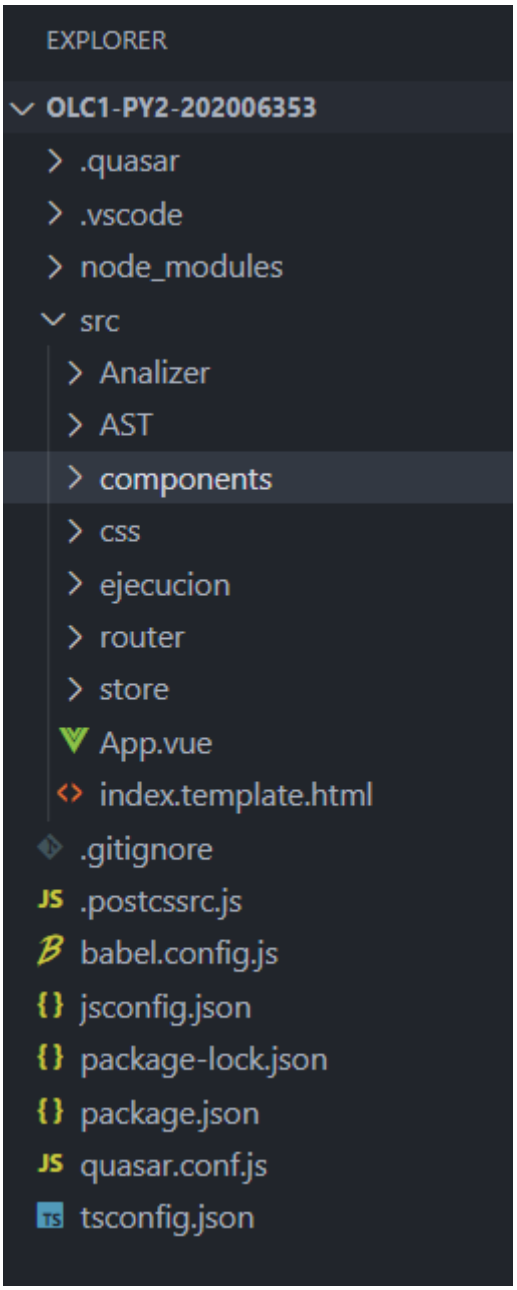
Requisitos Mínimos del Sistema

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)

Estructura raíz:

El proyecto tiene la siguiente estructura de directorios:



Directorio de la App: Raiz

La carpeta src:

Contiene todo el código utilizado para la implementación de este proyecto, dentro del mismo se encuentran varios directorios clave para el funcionamiento del proyecto, tal cual lo es el archivo de gramática, y el módulo para generación de reportes AST, estos módulos se describirán mas a detalle a continuación.

Analizer/gramática.jison:

En este archivo se encuentra descrita detalladamente la gramática utilizada para la comprensión del lenguaje solicitado, declaración de los tokens y pertenecientes al lenguajes así como la gramática que podrá reconocer los patrones de tokens, el tipo de gramática es case insensitive, lo que significa que ingnorara las mayúsculas o minúsculas tomando cadenas de texto como iguales.

```
/* Definición Léxica */
%lex

%options case-insensitive

%%

\s+                                     // espacios en blanco
"//".*                                  // comentario simple
[/][*][^*]*[*]+([^/*][^*]*[*]+)*[/]     // comentario multilíneas

//Palabras reservadas
'number'        return 'number';
'void'          return 'void';
'boolean'       return 'boolean';
'string'        return 'string';
'const'         return 'const';
'return'        return 'return';
```

En la parte del análisis sintáctico se tomo en cuenta la recursividad por la izquierda por lo cual se generaron bloques de instrucciones recursivos por la izquierda para poder llamar a cada uno de los métodos y funciones declaradas. También se utilizó el método del árbol, para poder generar el ast por medio de la gramática llenando así cada uno de los nodos.

```
%start INICIO

%%

INICIO
    : INSTRUCCIONES EOF { return new NodoAST({label: 'INICIO', hijos: [$1], linea: yylineno}); }
;


INSTRUCCIONES
    : INSTRUCCIONES INSTRUCCION  { $$ = new NodoAST({label: 'INSTRUCCIONES', hijos: [...$1.hijos, ...$2.hijos], linea: yylineno}); }
    | INSTRUCCION                { $$ = new NodoAST({label: 'INSTRUCCIONES', hijos: [...$1.hijos], linea: yylineno}); }
;

INSTRUCCION
    : DEC_VARIABLE      { $$ = new NodoAST({label: 'INSTRUCCION', hijos: [$1], linea: yylineno}); }
    | DEC_FUNCION       { $$ = new NodoAST({label: 'INSTRUCCION', hijos: [$1], linea: yylineno}); }
    | DEC_ARREGLOS      { $$ = new NodoAST({label: 'INSTRUCCION', hijos: [$1], linea: yylineno}); }
    | ASIGNACION        { $$ = new NodoAST({label: 'INSTRUCCION', hijos: [$1], linea: yylineno}); }
```

Ast/Error:

Contiene la Clase Error, la cual será la encargada de guardar todo los errores encontrados a lo largo del proyecto, estos errores pueden estar a nivel léxico o a nivel sintactico. Y serán almacenado en una lista de Errores a la cual podremos acceder en cualquier momento, y será accedida específicamente del lado del Frontend de la aplicación al momento de ser detectado un error.

```
export class Error {
  tipo: string;
  descripcion: string;
  linea: string;
  columna: string;

  constructor({ numero, tipo, descripcion, linea, columna }: { numero: string, tipo: string, descripcion: string, linea: string, columna: string }) {
    const valor = linea;
    const valor2 = columna;
    const valor3 = numero;
    Object.assign(this, {numero: valor3.toString(), tipo, descripcion, linea: valor.toString(), columna: valor2.toString()})
  }
}
```

AST/salida:

Se encargar de manejar las salidas a hacia el frontend, esta clase nos ayudara a recopilar la información que deseamos plasmar en la salida del frontend.

```
export class Salida {
  private static instance: Salida;
  lista: String[];

  private constructor() {
    this.lista = [];
  }

  public static getInstance(): Salida {
    if (!Salida.instance) {
      Salida.instance = new Salida();
    }
    return Salida.instance;
  }

  public push(linea: string): void {
    this.lista.push(linea);
  }

  public clear(): void{
    this.lista = [];
  }
}
```

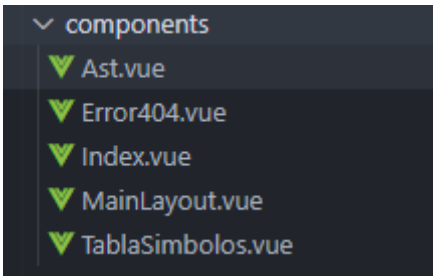
AST/nodoAST:

Esta clase se encarga de almacenar los nodos de los arboles, por medio de la cual podremos acceder a ellos y extraer la información que contenga cada uno, así como poder graficar dicho árbol por medio de la generación de un archivo con extensión .dot, archivo por medio del cual podremos generar el reporte de AST.

```
export const NodoAST = class NodoAST {
    constructor({ label = 'SIN ETIQUETA', hijos = [], linea = 0} = {}) {
        this.label = label;
        this.hijos = hijos;
        this.linea = linea;
    }
}
```

Componentes:

En este directorio podremos encontrar los archivos de extensión vue.js, por medio de los cuales fue creado el entorno grafico con la ayuda de quasar, el cual es el framework que se utilizó para generar el entorno gráfico y poder así representar correctamente el intérprete:



Como principal se podría mencionar el MainLayout, el cual es generado automáticamente por quasar como una plantilla par que se pueda editar, también podremos encontrar el archivo index.vue y dentro del cual podremos encontrar el diseño de la interfaz grafica del frontend, es acá donde se manda a llamar alas funciones que fueron antes declaradas.

```
<template>
  <q-page class="constrain q-pa-lg">

  <div>
    <q-splitter style=" padding-top: 10px; padding-left: 40px;">
      <template v-slot:after>
        <q-tabs v-model="tab" vertical style="width: 150px; height: 473px; background-color: black;">

          <q-btn-group push style="width: 150px; height: 45px; padding-top: 0px; background-color: black;">
            <q-btn label="OLC1_<202006353>" class="bg-black text-white" style="width: 150px;" disabled/>
          </q-btn-group>
```

Ejecucion:

En este directorio podremos enmcontrar todas las instrucciones que se generaran al momento de ejecutar codigo desde el frontend, podremos observar que hay muchas clases abstractas que pertenecen ala clase Instrucción, esto debido a que la clase instrucción es la raiz de la ejecucion, podremeos encontrar expresiones, asignaciones, declaraciones, de cada tipo de datos y tambien de cada tipo de funciones, todo tipo de asignaciones dentro de la clase abastracta por ejemplo:

La clase suma:

```
import { Entorno } from "../../entorno";
import { Instruccion } from "../../instruccion";

export class Suma extends Instruccion{
    expIzq: Instruccion;
    expDer: Instruccion;

    constructor(linea: string, expIzq: Instruccion, expDer: Instruccion){
        super(linea);
        Object.assign(this, {expIzq, expDer});
    }

    ejecutar(e: Entorno) {
        const exp1 = this.expIzq.ejecutar(e);
        const exp2 = this.expDer.ejecutar(e);

        return exp1 + exp2;
    }

}
```

Esta clase la podemos tomar como un prototipo del resto de las clases las cuales tambien heredan las propiedades de la clase abstracta Instrucción, por lo cual podremos usar los atributos iniciales d ela clase y por medio de la cual estaremos generando entornos.

La clase Entorno

Los entornos seran creados por medio de la clase Entorno, la cual se encargar de generar un Map por cada uno de los tipos de datos necesarios de los cuales dependera el resto de las clases:

```
import { Funcion } from './funcion';
import { Variable } from './variable';
import * as _ from 'lodash';

export class Entorno {
    variables: Map<String, Variable>;
    padre: Entorno;
    funciones: Map<String, Funcion>;

    constructor(padre?: Entorno) {
        this.padre = padre != null ? padre : null;
        this.variables = new Map();
        this.funciones = new Map();
    }
```

La clase Instrucción:

En esta clase encontraremos el origen de todo pues es desde esta clase que nacen todas las demas, la clase instrucción se encarga de abstraerse para poder generalizar todas las clases y que todas deriven o nazcan a partir de esta. En esta clase se definen los constructores de las clases abstractas y se manejan los atributos principales que contendran dentro de los entornos:

```
import { Entorno } from './entorno';

export abstract class Instruccion{
    linea: string;
    abstract ejecutar(e : Entorno) : any;

    constructor(linea: string){
        const valor = +linea + 1;
        Object.assign(this, {linea: valor.toString()});
    }

    getLinea() : string{
        return this.linea;
    }
}
```

Repositorio del proyecto: https://github.com/BeKingGO/OLC1-202006353