Universidad de San Carlos de Guatemala.

Facultad de ingeniería.

Nombre: JONATAN LEONEL GARCIA ARANA

Carné: 202000424

Sección: A

PROYECTO 1

MANUAL TECNICO

Introducción

El presente documento mostrara el manejo de una interfaz gráfica que por medio de un análisis léxico y sintáctico genera un árbol. Este manual será de gran utilidad para comprender el manejo del funcionamiento del programa y que requerimientos pide.

Información destacada

Este programa es único dado a base de los requerimientos y necesidades del proyecto que lo requiere.

Objetivos

Desarrollar una solución de software utilizando una interfaz gráfica que utilice técnicas de análisis léxico y sintáctico de un compilador para generar un análisis mediante el método del árbol. La solución deberá ser capaz de recibir un archivo de texto como entrada, analizar su contenido utilizando un conjunto de reglas gramaticales definidas previamente y generar un árbol que represente la estructura sintáctica del texto. La solución deberá ser fácil de usar y permitir la visualización del árbol generado para facilitar la comprensión del análisis sintáctico realizado.

Requerimientos

- Procesador de 32 bits (x86) o 64 bits (x64) a 1 gigahercio (GHz) o más.
- Memoria RAM de 1 gigabyte (GB) (32 bits) o memoria RAM de 2 GB (64 bits).
- Espacio disponible en disco rígido de 16 GB (32 bits) o 20 GB (64 bits).
- Dispositivo gráfico DirectX 9 con controlador WDDM 1.0 o superior.

Instalación y Configuración

Netbeans

Ingrese al siguiente enlace para descargar NetBeans 12.6
https://netbeans.apache.org/download/nb126/nb126.html

en esta página detalla todo el proceso de instalación.

INICIANDO EJECUCION DEL PROGRAMA

Clase Tree: en esta clase comenzamos con la creación del árbol, después de obtener la cadena en el cup la mandamos a este método el cual hace el Split de cada dato

```
public class Tree {

    node root;

    public Tree( String er, ArrayList<node> leaves, ArrayList<ArrayList> table ) {

        Stack pila = new Stack();
        String[] erSplit = er.split("");
        ArrayList<String> erSplitList = new ArrayList<>();
        ArrayList<String> erSplitList2 = new ArrayList<>();

        for (int i = 0; i < erSplit.length; i++) {
            if (erSplit[i].equals("|")) {
                erSplitList.add("|");
            } else if (erSplit[i].equals("*")) {
                erSplitList.add("*");
            } else if (erSplit[i].equals("+")) {
                erSplitList.add("+");
            } else if (erSplit[i].equals("?")) {
                erSplitList.add("?");
            } else if (erSplit[i].equals(".")) {
                erSplitList.add(".");
            } else if (erSplit[i].equals("#")) {
                erSplitList.add("#");
                erSplitList2.add("#");
            } else if (erSplit[i].equals("{")) {
                // Buscar el siguiente cierre de llave '}'
                int j = i + 1;
```

```
switch (character) {
    case "|":
        node lefto = (node) pila.pop();
        node righto = (node) pila.pop();

        node no = new node(character, Types.OR, 0, lefto, righto, leaves, table);
        pila.push(no);

        break;
    case ".":
        node lefta = (node) pila.pop();
        node righta = (node) pila.pop();

        node na = new node(character, Types.AND, 0, lefta, righta, leaves, table);
        pila.push(na);

        break;
    case "*":
        node unario = (node) pila.pop();

        node nk = new node(character, Types.KLEENE, 0, unario, null, leaves, table);
        pila.push(nk);

        break;
    case "+":
        node unario1 = (node) pila.pop();

        node nk1 = new node(character, Types.SUMA, 0, unario1, null, leaves, table);
        pila.push(nk1);
```

Clase para la tabla de transiciones

En esta clase creamos la tabla, la cual recibe la raíz del arbol y las hojas que se han creado en esta parte contamos con la validación del estado de aceptación de un estado y por cada estado que recorremos lo tenemos que ir buscando en el root para una mejor búsqueda

```
public transitionTable(node root, ArrayList tabla, ArrayList<node> leaves) {
    this.states = new ArrayList(); // estados del afd

    ArrayList datos = new ArrayList();
    datos.add("S0");
    datos.add(root.first);
    datos.add(new ArrayList());
    datos.add(false);

    int aceptacion = 0;

    this.states.add(datos);
    this.cont = 1;

    for (int i = 0; i < states.size(); i++) {
        ArrayList state = states.get(i);

        ArrayList<Integer> elementos = (ArrayList) state.get(1);
        Map<String, ArrayList<Integer>> transiciones = new HashMap<>();

        for (int hoja : elementos) {
            followTable sigTabla = new followTable();
            ArrayList lexemeNext = (ArrayList) sigTabla.next(hoja, tabla).clone();

            if (lexemeNext.get(0) == "") {
                continue;
            }
```

```
public void aceptacion(ArrayList<ArrayList> states, ArrayList<node> leaves) {
for(ArrayList state : states) {
    boolean acceptState = false;
    for(Integer leaf : (ArrayList<Integer>)state.get(1)) {
        if(new leave().isAccept(leaf, leaves)) {
            acceptState = true;
            break;
        }
    }
    state.set(3, acceptState);
}
```

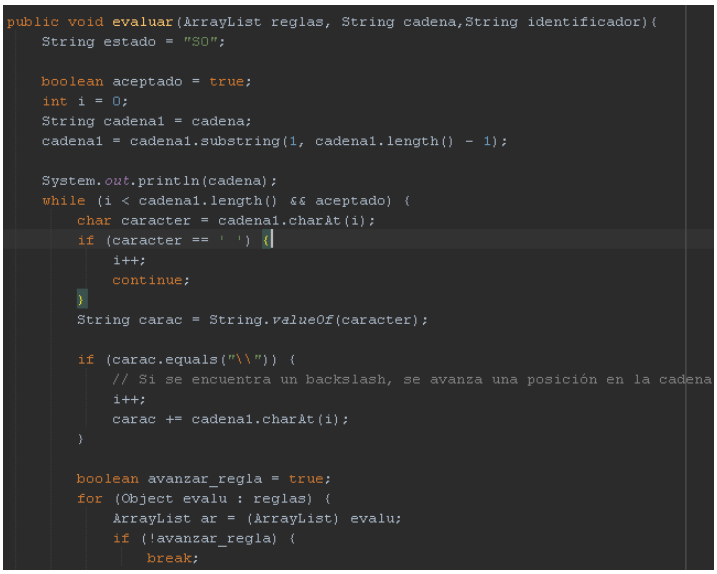
Método para evaluar cadena

En la primera parte de este método, mando a llamar la cadena que trae el archivo y a partir de la tabla de transición creo un array el cual contiene las reglas, esta se divide en dato que tiene que venir y estado al que tiene que pasar

```
public void evaluarcadena(String cadena, String nombre){
    int contador = 0;
      evaluar = new ArrayList();
     for(ArrayList state : states){
         System.out.println("-------------------------------");
          ArrayList<Object> dat = (ArrayList<Object>) state.get(2);
           evaluar2 = new ArrayList();
         for(Object state1: dat){

             String cadena1 = state1.toString();
             cadena1 = cadena1.replace("->", "");

             String[] palabras = cadena1.split("[\\s]+");

             String x = state.get(3).toString();
              ArrayList<String> sublista = new ArrayList<>(Arrays.asList(palabras));

              sublista.add(x);
              evaluar2.add(sublista);

         }
         evaluar.add(evaluar2);
         if (contador == states.size()-1){
             ArrayList<String> sublista1 = new ArrayList<>();
             String x1 = "S"+contador;
             sublista1.add(x1);
             sublista1.add("true");
             sublista1.add("true");
              sublista1.add("true");
```

Luego cuando ya tengo las reglas y la cadena mando a llamar otro método

Este método es el que hace el Split de la cadena y va evaluando según las reglas que hicimos anteriormente



```
public void evaluar(ArrayList reglas, String cadena,String identificador){
    String estado = "S0";

    boolean aceptado = true;
    int i = 0;
    String cadena1 = cadena;
    cadena1 = cadena1.substring(1, cadena1.length() - 1);

    System.out.println(cadena);
    while (i < cadena1.length() && aceptado) {
        char caracter = cadena1.charAt(i);
        if (caracter == ' ') {
            i++;
            continue;
        }
        String carac = String.valueOf(caracter);

        if (carac.equals("\\")) {
            // Si se encuentra un backslash, se avanza una posición en la cadena
            i++;
            carac += cadena1.charAt(i);
        }

        boolean avanzar_regla = true;
        for (Object evalu : reglas) {
            ArrayList ar = (ArrayList) evalu;
            if (!avanzar_regla) {
                break;
```

En este caso separa por carácter o por dato que pueda venir como son los caracteres especiales, tiene la lógica que si el carácter es igual a la regla en la que el array este sigue avanzando en la cadena si no es así, termina el proceso

Estructura del arbol

```
public node(String lexeme, Types type, int number, Object left, Object right, ArrayList<node> leaves, ArrayList<ArrayList> table) {
    first = new ArrayList();
    last = new ArrayList();
    anullable = true;

    this.lexeme = lexeme;
    this.type = type;
    this.number = number;

    accept = "#".equals(this.lexeme);

    this.left = left;
    this.right = right;

    this.leaves = leaves;
    this.table = table;
}
```

aquí se hace el método para ver si es anulable dependiendo los casos que son and, or, etc

```
 case HOJA:
     this.anullable = false;
     this.first.add(this.number);
     this.last.add(this.number);
     break;
 case AND:
     if(leftNode instanceof node && rightNode instanceof node){
         this.anullable = ((node)leftNode).anullable && ((node) rightNode).anullable;

         this.first.addAll(((node)leftNode).first);
         if(((node)leftNode).anullable){
             this.first.addAll(((node)rightNode).first);
         }

         if(((node)rightNode).anullable){
             this.last.addAll(((node)leftNode).last);
         }
         this.last.addAll(((node)rightNode).last);
     }
     break;
 case OR:
     if(leftNode instanceof node && rightNode instanceof node){
         this.anullable = ((node)leftNode).anullable || ((node) rightNode).anullable;

         this.first.addAll(((node)leftNode).first);
         this.first.addAll(((node)rightNode).first);


         this.last.addAll(((node)leftNode).last);
```

Método para obtener los siguientes

Aquí tenemos la lógica para encontrar los siguientes, kleene y suma tiene la misma lógica ya que en caso de kleene puede venir 0 o más veces y suma 1 o mas veces

```
public Object follow(){
    Object leftFollow=  this.left instanceof node ? ((node) this.left).follow() : null;
    Object rightFollow =  this.right instanceof node ? ((node) this.right).follow() : null;

    if(null != this.type)switch (this.type) {
        case AND:
            for (int item : ((node)leftFollow).last) {
                leave hoja = new leave();
                node nodo = hoja.getLeave(item, leaves);
                followTable tabla = new followTable();
                tabla.append(nodo.number, nodo.lexeme, ((node) rightFollow).first, table);
            }
            break;
        case KLEENE,SUMA:
            for (int item : ((node)leftFollow).last) {
                leave hoja = new leave();
                node nodo = hoja.getLeave(item, leaves);
                followTable tabla = new followTable();
                tabla.append(nodo.number, nodo.lexeme, ((node) leftFollow).first, table);
            }
            break;
        default:
            break;
    }

    return this;
}
```

Clase Errores en esta clase guardamos los errores que encontremos al analizar el archivo

```
public class Errores {
    static List<Errores> L_errores = new ArrayList();
    String lexema;
    int fila;
    int columna;
    String Tipo;
    String descripcion;

    Errores(String lexema,int fila, int columna, String Tipo, String descripcion){

        this.lexema = lexema;
        this.fila = fila;
        this.columna = columna;
        this.Tipo = Tipo;
        this.descripcion = descripcion;


    }

    static void addErrores(String lexema, int fila, int columna, String Tipo, String descripcion) {

        L_errores.add(new Errores((String) lexema,fila,columna,Tipo,descripcion));



    }
    static void ImprimirErrores(){
                for (Errores er : L_errores){
```

Clase resultados esta clase contiene los métodos para los resultados que obtuvimos del archivo

```
public class resultados {
    static List<Conjuntos> Conjun = new ArrayList();
    static List<expresiones1> Expresiones1 = new ArrayList();
        static List<Conjuntos2> Conjun2 = new ArrayList();
                static List<Oraciones> ORA = new ArrayList();


    public static void addconj(Object a,Object b) {
        //convertir a string
        String x = a.toString();
        String y = b.toString();
        String ySin = y.replaceAll("\\s+", "");
        Conjun.add(new Conjuntos(x,ySin));



    }
    public  static void addconj2(Object a,Object b) {
        //convertir a string
        String x = a.toString();
        String y = b.toString();
        String ySin = y.replaceAll("\\s+", "");
        Conjun2.add(new Conjuntos2(x,ySin));



    }
```