Manual Técnico (Compilador De Lenguajes)

Lenguaje: Java 11
Herramientas Secundarias: jflex, cup
Fecha De Desarrollo: Agosto-Septiembre 2020
Desarrollador:

Nombre: Sergio Daniel CIfuentes Argueta
GitHub: https://github.com/SergioCifuentes/
E-Mail: sergiodaniel-cifuentesargueta@cunoc.edu.gt

Para entender la estructura y funcionalidad del código que implementa el proyecto, es necesario describir el uso de las herramientas jflex y cup en la aplicación. Esas herramientas se utilizan en el análisis del archivo de entrada (.len) que representa el lenguaje que se desea cargar. En el archivo"editadordecodigo.jfle.lexicoLenguaje.jflex" demuestra los diferentes tokens que se utilizaran para la gramática, estos token se utilizan en el AnalizadorSintacticoLenguaje generado por cup, con el objetivo de construir una clase de tipo lenguaje. La gramática que utiliza el AnalizadorSintacticoLenguaje se describe en los archivos de GramaticaLenguaje.txt y ReglasSemanticas(Compilador De Lenguajes).pdf que se encuentran en la carpeta gramática. La base de esta gramática es la siguiente:

```
S: {Inicio}

INICIO        ::= INFO FUENTE  EXPRESIONES separador SIMBOLOS separador GRAMATICA
```

En dónde INFO representa un arreglo de String describiendo el lenguaje, FUENTE será el conjunto de líneas de código java que utilice el lenguaje, SÍMBOLOS es un ArrayList<Expresion> que son los símbolos utilizados en la gramática y GRAMTICA devolverá un ArrayList<Produccion> que representa las reglas gramáticas que se tendrá que seguir para cumplir el lenguaje.

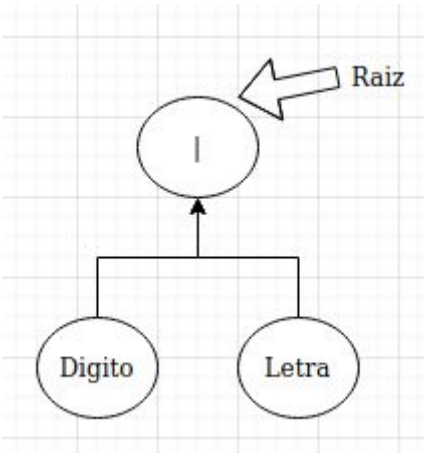
El objeto Expresión que se utilizara para construir el lenguaje como se vio previamente, se construye con los siguientes atributos

```
public Expresion(EstadoAVL raiz, String nombre) {
    numrosDeEstado= new ArrayList<>();
    amperson=false;
    this.raiz = raiz;
    this.nombre = nombre;
    agregarNumerosSinRepetir(obtenerNumeros(raiz));
}

public Expresion(EstadoAVL raiz, boolean amperson) {
    numrosDeEstado= new ArrayList<>();
    nombre=AMPERSON;
    this.raiz = raiz;
    this.amperson = amperson;
    obtenerNumeros(raiz);
    agregarNumerosSinRepetir(obtenerNumeros(raiz));
}
```

Existen dos constructores, uno para las expresiones que se deben ignorar representados por & y las expresiones normales con nombre; La instancia raíz del objeto Estado AVL es el primer Estado del árbol que genera la expresión regular, por ejemplo:

- Dígito|Letra



En esta caso el nodo "|" será la raíz de la expresión, teniendo como atributo el nodo1 (Dígito) y nodo2 (Letra).

Ahora el objeto Producción que también se utiliza en la construcción del objeto se forma de la siguiente manera.

```
public class Produccion {
    private Simbolo noTerminal;
    private ArrayList<Simbolo> producciones;
    private String code;

    public Produccion(Simbolo noTerminal, ArrayList<Simbolo> producciones, String code) {
        this.noTerminal = noTerminal;
        this.producciones = producciones;
        this.code = code;
    }
```

En donde el "noTerminal" representa el SÍmbolo que se deriva para obtener las "producciones" y el code es el código java que se ejecuta al momento de reducir la producción.

Construcción Del Analizador Sintactico:

Todos estos elementos previamente mencionados que genera el AnalizadorSintacticoLenguaje se llevan a una clase "ConstructorDeLenguaje" que construye el lenguaje de la siguiente manera:
Primero construye la TablaLALR con la clase ConstructorDeEstados con los símbolos y producciones que se le habilita. Para la construcción de la tabla se encuentran los primeros y siguientes de cada Simbolo no terminal con el objetivo de armar la tabla de primero y siguientes, luego se agrega una producción que sirva como comodín. que será Inicio'->Inicio$, esta se representa como una instancia final de Simbolo:

```
public  static final Simbolo COMODIN  =new Simbolo("$", "$comodin$", true);
```

Ya con estos datos ya estamos listos para empezar a derivar las producciones y crear los estados. Para esto utilizamos el siguiente método que se encuentra en el ConstructorDeEstados.

```
estados.add(cerradura(produccionEstados));
for (int i = 0; i < estados.size(); i++) {
    estados.get(i).setConecciones(irA(estados.get(i)));
    for (int j = 0; j < estados.get(i).getConecciones().size(); j++) {
        estados.get(i).getConecciones().get(j).setEstadoFinal(cerradura(estados.get(i).getConecciones().get(j).getProduccionesIniciales()));
        Estado existente = regresarEstadoIgual(estados.get(i).getConecciones().get(j).getEstadoFinal());
        if (existente == null) {
            estados.add(estados.get(i).getConecciones().get(j).getEstadoFinal());
        } else {
            estados.get(i).getConecciones().get(j).setEstadoFinal(existente);
        }

    }
}
```

El método cerradura genera un estado nuevo según la derivación que se hace con las producciones. Ya que la derivación recorre las conecciones de los producciones y verifica si es compatible con el símbolo que se evalúa.
Luego de tener los estados, se genera una tabla de “Accion” que es padre de clases como GoTo, Shift, Reduce, Aceptación. Estos conformarán el recorrido de la tabla para analizar sintácticamente el código engranaje.

Construcción Del Analizador Léxico:

Ahora para el Analizador Léxico que utiliza el programa, se utiliza un AFD que se debe de construir en base a las expresiones regulares que se habilitan en la entrada(. len). Todo esto lo administra la clase ControladorAfd, y para lograrlo se utiliza el metodo del arbol. Para construir el arbol las expresiones se juntan con una unión y luego se recorren para conseguir los nodos anulables, primeros y siguientes. Con los siguientes metodos recursivos:
Primeros:

```
public ArrayList<Integer> verificarPrimeros() {
    if (primeros.isEmpty()) {
        if (estado2 != null) {
            switch (operacion.tipo) {
                case Operacion.CONCATENACION:
                    if (estado1.anulable) {
                        primeros = agregarSinRepetir(estado1.verificarPrimeros(), estado2.verificarPrimeros());
                    } else {
                        primeros = estado1.verificarPrimeros();
                        estado2.verificarPrimeros();
                    }
                    return primeros;
                case Operacion.O:
                    primeros = agregarSinRepetir(estado1.verificarPrimeros(), estado2.verificarPrimeros());
                    return primeros;
                default:
                    throw new AssertionError();
            }
        } else if (operacion != null) {
            primeros = estado1.verificarPrimeros();
            return primeros;
        } else {
            primeros = estado1.verificarPrimeros();
            return primeros;
        }
    } else {
        return primeros;
    }
```

Siguientes:

```
public void obternerSiguientes(TablaDeSiguientes tabla) {
    if (operacion != null) {
        if (operacion.getTipo() == Operacion.CONCATENACION) {
            for (int i = 0; i < estado1.ultimos.size(); i++) {
                for (int j = 0; j < estado2.primeros.size(); j++) {
                    if (!tabla.getTable().get(estado1.ultimos.get(i)-1).contains(estado2.primeros.get(j))) {
                        tabla.getTable().get(estado1.ultimos.get(i)-1).add(estado2.primeros.get(j));
                        Collections.sort(tabla.getTable().get(estado1.ultimos.get(i)-1));
                    }
                }
            }
        } else if (operacion.getTipo() == Operacion.CERO_O_MAS_VECES||operacion.getTipo() == Operacion.UNA_O_MAS_VECES) {
            for (int i = 0; i < estado1.ultimos.size(); i++) {
                for (int j = 0; j < estado1.primeros.size(); j++) {
                    if (!tabla.getTable().get(estado1.ultimos.get(i)-1).contains(estado1.primeros.get(j))) {
                        tabla.getTable().get(estado1.ultimos.get(i)-1).add(estado1.primeros.get(j));
                        Collections.sort(tabla.getTable().get(estado1.ultimos.get(i)-1));
                    }
                }
            }
        }
```

Con estos datos generamos una tabla de transición que utilizara la pila de símbolos al momento de analizar el código entrada. Esta pila se conforma de la tabla, un ArrayList<Integer> que representan los estados que definirán los Shifts y GoTos, además de un arrayList<Símbolo> que representan los siglos de entrada que serán reducidas por las producciones.

Proceso de Lectura de Código:

Ya con el lenguaje construido por completo con la tablaLALR y tabla de transiciones léxico, es momento de empezar a recibir código de entrada. Para ello al momento de calcular se instancia un analizador Sintáctico con el parámetro de una instancia del analizador Léxico.

```
public void compilar() {
    indiceFinal = 0;
    AnalizadorLexico al = new AnalizadorLexico(lenguaje.getEstadoInical(), this);
    as = new AnalizadorSintactico(al, lenguaje, principal);
    as.start();

}
```

Esto funcionará de la siguiente manera: mientras no existan errores sintácticos este analizador le seguirá pidiendo tokens al analizador léxico mientras existan. Para mandar estos token el analizador utiliza un recorrido de la tabla de transiciones esto formará rutas mientras existan caminos para las expresiones y sino verifica si la ruta actual es terminal y si es así se envía ese token y si no se reduce en un paso hasta que se encuentre un estado terminal. Con este token el sintáctico recorre la tablaLALR realizando las acciones que se encuentre.

```
public void start(){
    obtenerEntrada();
     pila = new Pila(lenguaje.getTablaLR(),principal,entrada);
    while (true) {
        Token siguiente=obtenerSiguienteToken();
        obtenerEntrada();

        if (siguiente==null) {
            boolean listo = pila.ingresarSiguienteToken(null,entrada);
            if (!listo) {
                break;
            }
        }else{
            boolean listo = pila.ingresarSiguienteToken(siguiente,entrada);
            if (!listo) {
                break;
            }
        }

    }
```