Calculadora con expresiones regulares y gramáticas libres de contexto.

Autor:
Pérez Santiago.
sperez@comunidad.unnoba.edu.ar

Asignatura :
Ciencias de la computación 1

Docente:
Sabrina Pompei

Licenciatura en Sistemas

Escuela de Tecnología, Universidad Nacional del Noroeste de la Pcia de Bs As.

2022

Resumen. En este documento se busca explicar y desarrollar los conceptos vistos en la asignatura Ciencias de la Computación 1 y aplicarlos en la creación de una calculadora con operaciones aritméticas además de otras funcionalidades, se documentará el proceso que se debe realizar utilizando el lenguaje Java. Las herramientas para validar las entradas y los pasos que debe seguir para llegar a mostrar un resultado deseado se llevarán a cabo con el uso de las librerías JFlex para el análisis léxico y JCup para el análisis sintáctico. También se visualizarán capturas de la calculadora, se utilizará la herramienta JFLAP para representar algunos autómatas, expresiones regulares y gramaticas libres de contexto las cuales se utilizan para el desarrollo de la calculadora.

Palabras Clave: Java, JCUP, JFLEX, Graphviz, dot, automatas, expresiones regulares, gramáticas libres de contexto, calculadora, funciones.

1 Introducción.

En la actualidad las calculadoras son una herramienta que tenemos incorporadas en dispositivos celulares, en computadoras y en la red, además se encuentran también en diferentes páginas que proporcionan el servicio de una calculadora. Existen diferentes tipos como las básicas, financieras o científicas.
Como es un hecho y además ésta es utilizada por muchas personas para diferentes propósitos dándole diferentes usos, en este trabajo práctico se desea mostrar los pasos a la hora de programar una aplicación que funcione como una calculadora, para saber de qué forma realiza determinados procesos dentro de la lógica del proyecto. Dichos procesos van a llevarse a cabo con conceptos que se vieron en Ciencias de la Computación 1 y además se refleja qué reglas debe realizar para llegar al resultado final, o sea, en base a una entrada realice el cálculo y muestre el resultado.
Se documentará cómo se desarrolla una calculadora básica con operaciones aritméticas agregando otras funcionalidades como valor absoluto, potencia y raíz. Cabe aclarar que al ser una aplicación escalable se le pueden agregar nuevas funciones a futuro, pero obviamente dichas funciones se encuentran en cualquier otra calculadora en la red.

2 Teoría de Autómatas, Expresiones Regulares y Gramaticas Libres de contexto.

Antes que nada, para mejor entendimiento del uso de las herramientas es bueno explicar los conceptos teóricos de los autómatas, las expresiones regulares y gramáticas libres de contexto.

2.1 Autómatas

Un autómata es un modelo matemático para una máquina de estado finito. Dada una entrada de símbolos, "salta" a través de una serie de estados de acuerdo a una función de transición (que puede ser expresada como una tabla). En la variedad común "Mealy"[1] de FSMs, esta función de transición dice al autómata a qué estado cambiar dados unos determinados estado y símbolo. [2]

Los Autómatas Finitos son máquinas teóricas que van cambiando de estado dependiendo de la entrada que reciban. La salida de estos autómatas está limitada a dos valores: **aceptado y no aceptado**, que pueden indicar si la cadena que se ha recibido como entrada es o no válida. Generalmente se utilizan para reconocer lenguajes regulares, es decir, una palabra se considerará válida sólo si pertenece a un determinado lenguaje.

Formalmente, un Autómata Finito (AF) se define como una tupla

$$AF = (\Sigma, Q, f, q0, F), \text{ donde}$$

- Σ es el alfabeto de entrada
- Q es el conjunto finito y no vacío de los estados del Autómata
- f es la función de transición que indica en qué situaciones el Autómata pasa de un estado a otro, se define $f : Q \times \Sigma \rightarrow Q$
- $q0 \in Q$ es el estado inicial
- $F \subset Q$ es el conjunto de estados finales de aceptación ($F \neq \emptyset$) [3].

En el Autómata finito determinista (AFD) cada estado de un autómata de este tipo puede o no tener una transición por cada símbolo del alfabeto.

Un ejemplo podemos ver en la Figura 1.

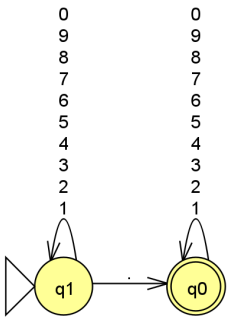

Fig. 1. Autómata que reconoce números flotantes.

También existe el Autómata finito no determinista (AFND) donde sus estados pueden, o no, tener una o más transiciones por cada símbolo del alfabeto. El autómata acepta una palabra si existe al menos un camino desde el estado q_0 a un estado final F etiquetado con la palabra de entrada. Si una transición no está definida, de manera que el autómata no puede saber como continuar leyendo la entrada, la palabra es rechazada.

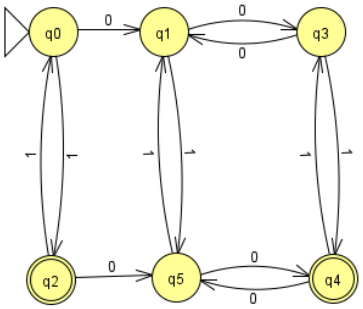

Fig. 2. Autómata que reconoce cantidades pares de 0 e impares de 1.

El Autómata finito no determinista con transiciones ε (AFND-ε) además de ser capaz de alcanzar más estados leyendo un símbolo, permite alcanzarlos sin leer ningún símbolo. Si un estado tiene transiciones etiquetadas con ϵ, entonces el AFND puede encontrarse en cualquier de los estados alcanzables por las transiciones ϵ, directamente o a través de otros estados con transiciones ϵ .
El conjunto de estados que pueden ser alcanzados mediante este método desde un estado q, se denomina la clausura ϵ de q.

Sin embargo, todos estos tipos de autómatas pueden aceptar los mismos lenguajes. Siempre se puede construir un AFD que acepte el mismo lenguaje que el dado por un AFND. [3]

2.2 Expresiones Regulares.

Una expresión regular es una notación normalizada para representar lenguajes regulares que permiten describir con exactitud y sencillez cualquier lenguaje regular. Para definir una ER se pueden utilizar todos los símbolos del alfabeto Σ y, además, λ y Ø.

Los operadores que también se pueden utilizar son:

- + representa la unión.
- . representa la concatenación (este símbolo no se suele escribir).
- * representa la clausura de Kleene.[4]
- ( ) modifican las prioridades de los demás operadores.

Una expresión regular se puede definir de acuerdo a los siguientes criterios:

1. Ø es una e.r. que representa al lenguaje vacío (no tiene palabras) $L_{\emptyset} = \emptyset$
2. λ es una e.r. que representa al lenguaje $L_{\lambda} = \{\lambda\}$
3. $a \in \Sigma$ es una e.r. que representa al lenguaje $L_a = \{a\}$
4. Si α y β son e.r. entonces $\alpha + \beta$ también lo es y representa al lenguaje $L_{\alpha+\beta} = L_\alpha \cup L_\beta$
5. Si α y β son e.r. entonces $\alpha\beta$ también lo es y representa al lenguaje $L_{\alpha\beta} = L_\alpha L_\beta$
6. Si α es una e.r. entonces α^* también lo es y representa al lenguaje $L_{\alpha^*} = \bigcup_{i=0}^{\infty} L_\alpha^i$ que también se puede representar $\alpha^* = \Sigma_{i=0}^{\infty} \alpha^i$

Fig. 3. Criterios ER.

Solo son ER. aquellas que puedan ser definidas utilizando los 6 puntos de la figura 3.
La prioridad de las operaciones, que puede modificarse utilizando paréntesis, es de mayor a menor: * . + [5]

Un ejemplo de expresiones regulares puede ser (101)* donde solamente aceptará cadenas del tipo 101,101101,101101101 o simplemente una cadena vacía (λ).

2.3 Gramáticas Libres de Contexto

Una gramática libre de contexto es una gramática formal en la que cada regla de producción es de la forma:

V -> w

Donde V es un símbolo no terminal y w es una cadena de terminales y/o no terminales. El término libre de contexto se refiere al hecho de que el no terminal V puede siempre ser sustituido por w sin tener en cuenta el contexto en el que ocurra. Un lenguaje formal es libre de contexto si hay una gramática libre de contexto que lo genera.
Las gramáticas libres de contexto permiten describir la mayoría de los lenguajes de programación, de hecho, la sintaxis de la mayoría de lenguajes de programación está definida mediante gramáticas libres de contexto. Por otro lado, estas gramáticas son suficientemente simples como para permitir el diseño de eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada, determinen cómo puede ser generada desde la gramática. Los analizadores LL y LR tratan restringidos subconjuntos de gramáticas libres de contexto.
La notación más frecuentemente utilizada para expresar gramáticas libres de contexto es la forma Backus-Naur.[6]

Una gramática libre de contexto puede ser definida mediante la 4-tupla:

$G = (V_t, V_n, P, S)$ donde

- V_t es un conjunto finito de terminales
- V_n es un conjunto finito de no terminales
- P es un conjunto finito de producciones
- $S \in V_n$ el denominado Símbolo Inicial
- los elementos de P son de la forma

$$V_n \longrightarrow (V_t \cup V_n)^*$$

donde * denota el operador estrella de Kleene.

Fig. 4. Definición formal GLC.

Podemos dar un ejemplo de gramática libre de contexto con las reglas que seguirá la calculadora posteriormente:

1) R -> E.
2) E -> E + T | E - T | T.
3) T -> T * MU | T / MU | MU.
4) MU -> - MU | F.
5) F -> int | float | (E) | FA.
6) FA -> abs (E) | pot (E comma E) | raíz (E comma E).

Donde R = resultado, E = Expresión, T = Término, MU = menos unario, F = Factor, FA = función adicional.

3 Implementación de la calculadora.

Una vez visto los conceptos teóricos procedemos a explicar la manera en como se pensarán los procesos del desarrollo de la calculadora. Podemos decir que en todo proyecto se debe tener una forma organizada por eso en la punto 3.2, donde se explica la estructura que se adopta el proyecto que se puede adaptar para nuevos proyectos. También, se debe especificar el lenguaje ya que no es lo mismo la forma de implementar código en el lenguaje Java que en el lenguaje de Python.

Debemos tener en cuenta, además del lenguaje, qué herramientas nos servirán para llevar a cabo la implementación.

Por último mencionar que el desarrollo de la calculadora se pensó en 4 pasos para su desarrollo:

- Análisis Léxico de la calculadora: Reconocimiento de entradas y validaciones.
- Análisis Sintáctico de la calculadora: Las entradas deben cumplir con las reglas sintácticas de una calculadora.
- AST de una entrada: Muestra simplificada del recorrido por las reglas.
- Cálculo y muestra del resultado.

3.1 Lenguaje elegido.

Es este trabajo por preferencia, se eligió el lenguaje de programación Java.

Java es una plataforma informática de lenguaje de programación, es una plataforma fiable en la que se crean muchos servicios y aplicaciones, también productos y servicios digitales diseñados para el futuro también se programan en Java.[7]

Con una definición del lenguaje también podemos aclarar que existen otros lenguajes para poder desarrollarlo como Python, C o C++, cada una con su propia forma de llevarlo a cabo.

3.2 Estructura del Proyecto.

Como se habló antes, se necesita una organización, una estructura del proyecto para entender los pasos que va a realizar el código.

A continuación se muestra la estructura del proyecto posicionándonos en la carpeta raíz.

```
|CARPETA
|   arbol.dot | arbol.png | output.txt | resultado.txt
+---CalculadoraCiencias1
|   |   pruebas.txt
|   +---Calculadora
|   |   \---src
|   |       +---archivos
|   |       |   |   AbrirParteGráfica.java | GenerarDOT.java | GenerarFlexyCup.java
|   |       |   +---ast
|   |       |   |   \---Base
|   |       |   |       |   Nodo.java | Resultado.java
|   |       |   |       \---Expresiones
|   |       |   |           |   Expresion.java
|   |       |   |           +---Constantes
|   |       |   |           |       Constante.java | ConstanteEntera.java | ConstanteFloat.java
|   |       |   |           +---funcionesAdicionales
|   |       |   |           |       ABS.java | POTENCIA.java | RAIZ.java
|   |       |   |           \---Operaciones
|   |       |   |               +---binarias
|   |       |   |               |   |   OperacionBinaria.java
|   |       |   |               |   \---arismeticos
|   |       |   |               |           Division.java | Multiplicacion.java | Resta.java | Suma.java
|   |       |   |               \---unarias
|   |       |   |                       OperacionUnaria.java | MenosUnario.java
|   |       |   +---lexico
|   |       |   |       EjemploJFlex.java | lexico.flex | MiLexico.java | MiLexico.java~ | MiToken.java
|   |       |   \---sintactico
|   |       |           EjemploJavaCup.java | MiParser.java | MiParserSym.java | parser.cup
|   +---librerias
|   |       java-cup-11b-runtime.jar | java-cup-11b.jar | jflex-full-1.8.2.jar
```

Fig. 5. Estructura del Proyecto.

En la figura 5 se muestran los archivos relevantes que posee el proyecto, otras carpetas y archivos se comprenden a propios del IDE, en este caso IntelliJ, se omiten.

Podemos dar una pequeña descripción de lo que tiene cada carpeta.

CARPETA: es la carpeta raíz que contiene todos las demás carpetas y archivos, posee 4 archivos que posteriormente se dirá su uso.

CalculadoraCiencias1: contiene el archivo pruebas.txt que es donde se almacena la entrada de los valores.

Calculadora: contiene toda la lógica del proyecto.

librerias: tiene las librerias que se usan en el proyecto

archivos: tiene los 3 archivos con extensión .java que sirven de generadores del lexer, parsing, ast y obviamente el resultado.

ast: contiene todo el diagrama de clases que servirá además de dibujar el ast, también hacer la lógica de las operaciones aritméticas y las funciones adicionales, sus carpetas son para mantener una organización del proyecto.

lexico: contiene la implementación del lexer, y archivos que nos ayudan a correr el lexer.

sintactico: contiene la implementación encargada de realizar las acciones sintácticas y el recorrido por las reglas. También se encarga de instanciar las clases de la carpeta ast.

3.3 Librerías

Para hacer uso de las librerías primero veamos una simple definición de las mismas. Una librería se puede entender como un conjunto de clases que facilitan operaciones y tareas ofreciendo al programador funcionalidad ya implementada y lista para ser usada. Nos permiten reutilizar código, es decir, podemos hacer uso de las clases, métodos y atributos que componen la librería evitando así tener que implementar nosotros mismos esas funcionalidades. [8]

3.3.1 JFLEX

JFLex es un metacompilador que permite la rápida generación de analizadores lexicográficos, desarrollado en Java, para Java, el cual posee una tecnología basada en autómatas finitos determinConstanteEntera constanteEntera

istas (AFD).[9]

Genera un analizador léxico para un lenguaje determinado (en nuestro caso, una calculadora). Se caracteriza porque utiliza expresiones regulares (regex) para reconocer los lexemas del lenguaje.

Algunas de las características más sobresalientes de JFLex son:

- Generación rápida de analizadores léxicos.
- Sintaxis cómoda de manipular y fácil de interpretar.
- Independencia de plataforma, ya que está generado para ser integrado con Java.
- Fácil integración con CUP.
- Es Open Source.

3.3.1.1 Estructura de un archivo JFLEX.

JFLex, necesita un archivo de configuración, en donde se especifique la descripción de la estructura de los tokens o expresiones regulares, para obtener la validación de los tokens en el analizador léxico. Las especificaciones o reglas, son escritas en un archivo con extensión .flex, el cual lee JFLex e integra con una clase 'Scanner.java' la cual permite analizar los tokens enviados y producir un programa llamado Yylex (en nuestro caso MiLexer.java) que reconoce las cadenas que cumplan las reglas establecidas para posteriormente ejecutar las acciones asociadas.

Un archivo JFlex está formado por cinco bloques separados por los símbolos '%%':

1) Código, importaciones y paquete.
2) Directivas JFLex (opciones y declaraciones)
    a) Opciones
    b) Declaraciones
3) Reglas

En el primer bloque deben encontrarse sentencias Java de importación de los paquetes que se vayan a utilizar en las acciones regulares situadas al lado de cada patrón en la zona de reglas, como también escribir clases completas e interfaces. Por ejemplo:

```
package archivos.lexico;
```

En el segundo bloque se le da configuración al archivo con parámetros que le dan comportamiento al archivo.

A continuación los parámetros :

- **%public:** genera una clase pública.
- **%final:** genera una clase final (sin herencia)
- **%int:** modifica la función de análisis yylex() la cual devolverá valores de tipo int en lugar de Yytoken

- **%eofclose:** provoca que yylex() finalice la lectura en cuanto encuentre EOF.
- **%unicode:** la entrada estará formada por caracteres Unicode.
- **%line:** guarda en la variable yyline el número de línea en que comienza el lexema actual. Puede consultarse con this.yyline
- **%class Nombre:** Nombre de la clase generada
- **%type Nombre:** Nombre de la clase usada para representar token. Tipo de los objetos retornados por yylex()
- **%column:** Número de columna que está analizando. Puede consultarse con this.yycolumn
- **%char:** Número de carácter que está analizando. Puede consultarse con this.yychar

Podemos dar el siguiente ejemplo:

```
%public
%class MiLexico
%unicode
%cup
%type MiToken
%line
%column
```

También puede incluir sus propias declaraciones y métodos en el interior de la clase generada, escribiendo a éstas entre los símbolos %{ y %}.
En esta sección se puede incluir código que se copiará textualmente como parte de la definición de la clase del analizador léxico. Típicamente serán variables de instancia o nuevos métodos de la clase.

En la parte de declaraciones las expresiones regulares se pueden definir como patrones que tienen un lenguaje propio y describen el formato del lexema. Los patrones definidos podrán ser utilizados posteriormente en cualquier otra regla encerrándolo entre llaves.
Por ejemplo : digito = [0-9] ; carácter = [a-zA-Z].

Y por último en la tercera y última parte de la especificación, contiene todos los patrones necesarios, utilizando expresiones regulares, para el realizar el análisis léxico a través de las acciones semánticas.

Por ejemplo:

```
%%
<YYINITIAL> {
 /* "operadores" */
 "+" {return token("SUMA", yytext()); }
 "-" {return token("RESTA", yytext());}
 "*" {return token("MULT", yytext());}
 "/" {return token("DIV", yytext());}
}
```

Por defecto, el estado activo es YYINITIAL, para activar otro estado, desde el inicial o un estado intermedio, se debe invocar, desde las acciones de las reglas, al método yybegin().

3.3.2 JCUP

CUP es una base para la construcción de analizadores sintácticos, obtiene su nombre por sus siglas en inglés Based Constructor of Useful Parsers. CUP es un sistema que permite la generación de analizadores o parsers de tipo LALR (Look – Ahead Left to Right parser) para Java. Genera un analizador sintáctico para una gramática determinada utilizando el método de parsing ascendente con código de producción y asociación de fragmentos de código JAVA. Cuando una producción en particular es reconocida, se genera un archivo fuente Java, MiParser.java que contiene una clase lr_parser, o sea, parsing ascendente.

Una de las grandes ventajas que existen en CUP, además de su velocidad, del gran número de nuevas características que presenta y su eficiencia en implementación al trabajar en conjunto con Java, CUP permite la integración de usuarios de otros metacompiladores como JFLEX.

3.3.2.1 Estructura de un archivo CUP.

Un archivo Cup está formado por cinco bloques:

- Definición de paquete y sentencias import.
- Código de usuario.
- Listas de símbolos.
- Reglas de asociatividad y precedencia.
- Gramática.

En el primer bloque definimos el o los paquetes del proyecto y las importaciones de clases o archivos.

Por ejemplo:

```
package archivos.sintactico;
import archivos.ast.Base.Nodo;
import archivos.ast.Base.Resultado;
import java_cup.runtime.Symbol;
import java.util.*;
```

En el segundo bloque se puede incluir código Java que el usuario desee incorporar en el analizador sintáctico que se va a obtener con CUP. Existen varias partes del analizador sintáctico generado con CUP en donde se puede incluir código de usuario, cada una tiene diferente función.

- action code {: código :}
  Dentro de esta función, el usuario agrega código propio, el cual, posteriormente, es agregado a una clase no pública para ser incluido a la clase del analizador, de esta forma es posible declarar variables o rutinas.
- parser code {: código:}
  Permite declarar métodos y variables que serán integrados directamente a la clase del analizador sintáctico, y aunque no sea una acción muy común, es posible agregar un método de análisis lexicográfico o alguna otra rutina. Por ejemplo:

```
parser code
{:
public String reglas = "";
public void concat_rules(String regla){reglas+=regla + "\n";}
:};
```

- init with {: código :}
  Proporciona un código que será ejecutado por el analizador antes de que solicite el primer token.
  Típicamente, esto se usa para inicializar el analizador léxico, así como varias tablas y otras estructuras de datos que puedan necesitar las acciones.
- scan with {: código :}
  Define como el analizador sintáctico debería verificar o validar sobre el siguiente token a leer, esta subsección, al Al igual que la anterior, posee un valor de retorno, pero a diferencia del anterior, retorna un objeto del tipo java_cup.runtime.Symbol.

En nuestro caso solamente utilizaremos la sección parser code.

La lista de símbolos es requerida ya que posee la declaración de los símbolos pertenecientes a la gramática. Dentro de esta sección, se declara la nomenclatura o nombramiento de cada símbolo terminal y no-terminal. También debe declararse el tipo, por defecto se declara tipo Object.

Por ejemplo:

```
terminal   SUMA,COMA,   RESTA,   MULT,   DIV,   PARENTESISO,
PARENTESISC;
nonterminal  Resultado resultado;
```

La sección de reglas de asociatividad y precedencia es opcional y funcional con gramáticas ambiguas permitiendo definir la precedencia y cuál será la asociatividad por cada declaración, CUP permite tres formas:

- precedence left terminal…;
- precedence right terminal…];
- precedence nonassoc terminal…;

En nuestro caso no se utiliza porque la precedencia y asociatividad se resuelve en la gramática de la calculadora.

En la sección de gramática es la última del fichero de configuración de CUP, generalmente empieza con la sección

```
start with non-terminal;
```

Siempre de forma opcional. De esta manera se indica cual no-terminal es el comienzo para el analizador sintáctico.

Se especifican además las reglas de producción que tienen las siguientes características:

- La gramática puede comenzar con una declaración que diga cuál es el símbolo inicial.
- Para definir un no terminal se utiliza el símbolo '::='.
- Las reglas finalizan con ';'.
- Las reglas de producción pueden incluir acciones semánticas, que son delimitadas por los símbolos '{:' y ':}'.
- El atributo del antecedente se llama RESULT, mientras que los atributos de los símbolos del consecuente son accesibles mediante los alias que el desarrollador haya indicado para cada uno de ellos. Se indica con un ':'.

Por ejemplo:

```
start with resultado;  se declara que inicie en resultado
resultado ::= expresion:e  e es un alias
    {:   concat_rules("REGLA 0: resultado --> expresion" +
"\n  --> " + e.darResultado()); acciones semanticas
   RESULT = new  Resultado("Resultado",e);
              atributo del antecedente
    :};
expresion ::=
   expresion:e1 SUMA termino:e2 {:
         concat_rules("REGLA 1.1: expresion --> expresion
SUMA termino " + "\n  --> " + e1.darResultado() +" + "+
e2.darResultado());
       RESULT = new Suma("Suma",e1,e2);
    :};
```

Notas importantes.

- **Cup debe implementar la interface java_cup.runtime.Scanner**
- **Se debe incluir %cup en la sección de directivas del archivo .flex.**
- **El analizador léxico debe devolver los tokens al analizador sintáctico en forma de objetos tipo MiToken que extiende de Symbol.Esta clase, que se encuentra en el paquete java_cup.runtime y que, por tanto, debe ser importada por el léxico.**

3.3.3 GRAPHVIZ

Graphviz es un conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT. [10]

DOT es un lenguaje descriptivo en texto plano. Proporciona una forma simple de describir grafos entendible por humanos y computadoras. Los ficheros de DOT suelen usar la extensión .gv o .dot.

Esta librería nos ayudará a dibujar un gráfico que muestre de forma simplificada el parsing que realiza nuestra calculadora.

Ejemplo lenguaje DOT:

```
graph G {nodo_resultado[label="Resultado 4920.0"]
nodo_1493365641[label=" - [ Suma ] - "]
nodo_resultado--nodo_1493365641
nodo_229883386[label=" - [ Multiplicacion ] - "]
```

Cerrando con las librerías podemos decir que su uso en esta aplicación y en muchas nos facilita mucho el trabajo a la hora de desarrollar.

3.4 Léxico

En esta etapa del proceso de desarrollo de la aplicación que tiene como servicio la calculadora buscamos primero definir las “palabras clave” que tiene una calculadora, en este caso vamos a hablar de Tokens que son una secuencia de caracteres que nuestro analizador léxico agrupa en una unidad representativa. Cada token tiene asociado un valor el cual viene dado por la secuencia de caracteres que lo generó llamado lexema.

Se programan autómatas de estado finito, pero es preferible escribirlos como expresiones regulares.

En nuestra calculadora podemos hacer referencia a eso cuando declaramos una asignación que contiene una expresión regular y dicha expresión regular se la asigna a un identificador, de la siguiente manera:

```
Int= \d+
```

Esta asignación tiene a Int como el identificador que posteriormente en el sector de reglas utilizaremos para instanciar un token.
Por otro lado, tenemos la expresión regular \d+, otra forma de representarla sería [0-9]*, significa que el identificador Int reconocerá solamente números enteros positivos. Podemos ver el autómata en la siguiente figura:

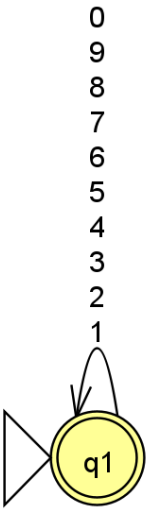

Fig. 6. Autómata que reconoce números enteros positivos.

En el código en la parte del inicio de las reglas y posterior a la etiqueta `<YYINITIAL>` podemos declarar que cuando se reconoce Int instanciamos un token. Ese token tiene tipo MiToken, que extiende de Symbol que a su vez es la clase para representar los tokens. Se declara en el sector de declaraciones con el siguiente constructor.

```
private MiToken token(String nombre, Object valor)
{return new MiToken(nombre, this.yyline, this.yycolumn,
valor);}
```

y después en el sector de reglas lo declaramos de la siguiente manera:

```
{Int} {
   int a = yytext().length();
   if ( a < cota_int ) {return token("INT", yytext());}
   else{
       return token("ERROR", "Supera cantidad máxima de
caracteres permitidos");
   }
}
```

Previamente en el sector de declaraciones se agregó una variable cota_int que nos sirve para limitar la cantidad de caracteres del lexema de Int.

```
int cota_int = 20;
```

En esta parte del código podemos analizar dos partes. Primero podemos ver que se fija la cantidad de caracteres que tiene yytext(), yytext() explicado brevemente sería el lexema.
En caso de que sea menor se instancia un nuevo token que tiene como nombre "INT" y el valor del lexema. De otra forma, se instancia también un token pero este token se llama "ERROR" y muestra de forma personalizada el error que tiene.
Esto nos sirve para que Int solamente se limite a 20 caracteres.

En la clase MiToken se sobreescribe la función toString().

```
@Override
public String toString() {
   String posicion = "";
   if (this.linea != -1 && this.columna != -1)
       posicion = " @ (L:" + this.linea + ", C:" +
this.columna + ")";
   if (valor == null)
       return "[" + this.nombre + "]" + posicion;
   else
       return "[" + this.nombre + "] -> (" + this.valor
+ ")" + posicion;
}
```

Podemos observar, que dependiendo del constructor que hayamos invocado, la forma que se representará el token.

A continuación en las siguientes figuras mostraremos los dos casos.

Caso 1:
Entrada:5000000000000000000000000000

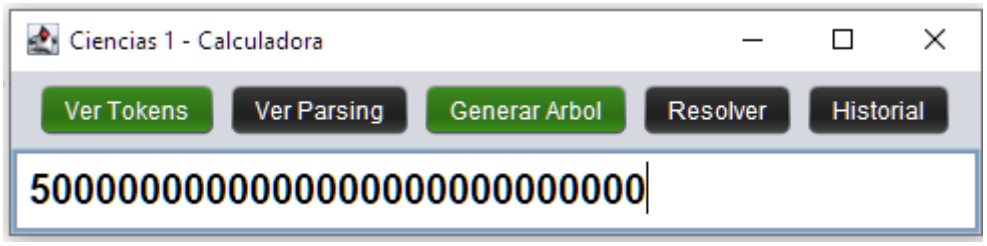

Fig. 7. Entrada para error.

Apretamos el botón para ver los tokens y nos muestra lo siguiente:

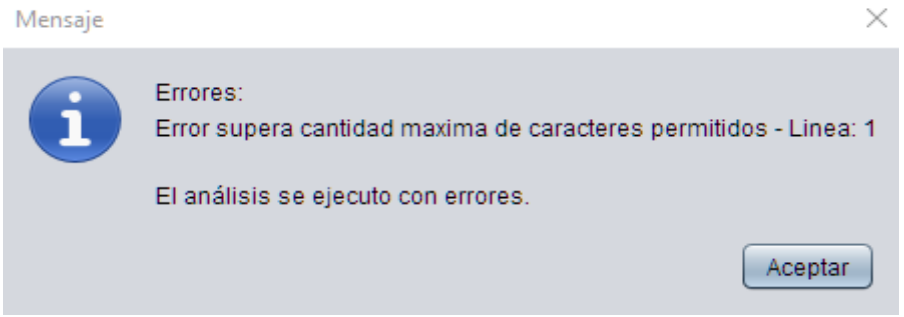

Fig. 8. Error en pantalla.

Caso 2:
Entrada:200

Fig. 9. Entrada sin error.

Para el resultado del token nos muestra lo siguiente:

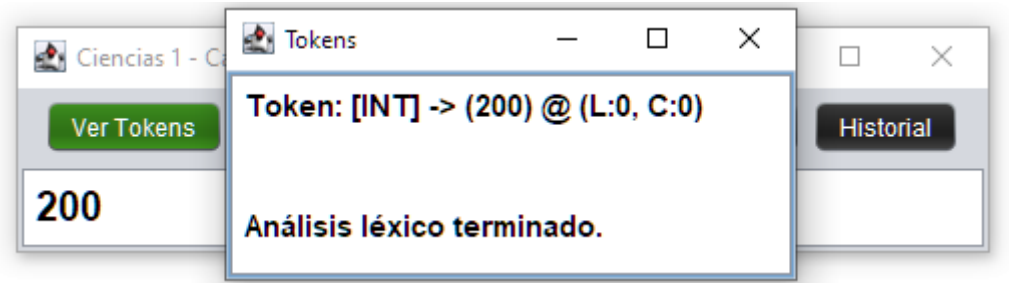

Fig. 10. Entrada sin error.

Con una simple muestra ahora proceder a explicar las siguientes reglas que tiene el analizador léxico.
Como hablamos de una calculadora, además de números enteros también tenemos números con decimal, en nuestro caso los llamamos float.
Los float tienen esta declaración:

```
Float = \d+\.\d+
```

Podemos ver que tiene que tener un punto entremedio de dos enteros.
Su cota es de 25 y se realiza la verificación de la misma manera que en caso del INT.

También están las declaraciones para las funciones de valor absoluto, potencia y raíz.

```
Potencia = "potencia"|"POTENCIA"
Raiz = "raiz"|"RAIZ"
Abs = "abs"|"ABS"
```

Estas nos ayudarán para posteriormente declarar las funciones adicionales, en éste caso podemos declarar el token directamente en el sector de reglas o de la siguiente manera.

```
{Potencia} {return token("POTENCIA", yytext());}
{Raiz} {return token("RAIZ", yytext());}
{Abs} {return token("ABS",yytext());}
```

Podemos observar que el identificador entre llaves {id}, nos reconoce como las declaraciones que se pusieron en su sector.

También podemos hacerlo directamente, este es el caso de los paréntesis y la coma.

```
"\," {return token("COMA", yytext());}
"\(" {return token("PARENTESISO", yytext());}
"\)" {return token("PARENTESISC", yytext());}
```

Como se observa, se debe agregar una barra "\" para declararlos, lo mismo sucede con las expresiones regulares de los enteros y float anteriormente vistos ya que representan un valor literal.

Terminando la parte del léxico, podemos ver las siguientes declaraciones.

```
LineTerminator = \r|\n|\r\n
WhiteSpace     =  \s | {LineTerminator} | [\t\f]
```

En ellas se observan caracteres literales, cada uno representa un caracter:

- \r Caracter de escape dentro de una cadena de caracteres que significa retorno de carro (carriage return).
- \f Coincide con un avance de página.
- \n Caracter de línea nueva.
- \s Caracter de espacio en blanco.

Cuando establecemos la regla de WhiteSpace no hacemos nada, básicamente estaríamos ignorando los espacios en blanco de la siguiente manera.

```
{WhiteSpace} { /* ignore */ }
```

Ya por último, como no tenemos nada más para agregar, terminamos con la sentencia regla [^], esta regla JFLEX la define como un error, por ende si lo que pusimos para analizar no es nada de las reglas anteriores, generará un error de la siguiente manera:

```
[^]{    return    token("ERROR",    "Illegal    character
<"+yytext()+">"); }
```

Podemos ver el siguiente caso

Entrada: hola mundo

Fig. 11. Entrada sin error.

Obtendremos lo siguiente:

Fig. 12. Error illegal character.

Ya una vez terminando esta parte podemos ver que agregando las siguientes reglas reconocemos las posibles entradas de una calculadora.

```
/* "operadores" */
"+" {return token("SUMA", yytext());}
"-" {return token("RESTA", yytext());}
"*" {return token("MULT", yytext());}
"/" {return token("DIV", yytext());}
```

Ahora podemos simular la entrada de una calculadora, generando el token y su lexema asociado que corresponde a las entradas que analizamos.

Entrada: -(50+30) * (potencia(50,2))

Tokens y lexemas generados:

Fig. 13. Tokens generados.

En la figura 13 se observa también la línea, que en nuestro caso es 1, y la columna que es donde detecta el lexema.

De esta parte podemos decir que el uso de las expresiones regulares para reconocer las entradas nos aporta mucho ya que es la primera verificación para el funcionamiento de la calculadora.

3.5 Sintáctico

En el análisis sintáctico lo que hace es utilizar una lista de tokens que se generan en la parte del análisis léxico, se los analiza aplicando reglas de gramáticas libres de contexto y devolverá una lista con las reglas que se fueron recorriendo, claro que si la lista de tokens no corresponde con las reglas de la gramática libre de contexto deberá arrojar un error de sintactico.

Con esto en mente podemos empezar a explicar en proceso de implementación del archivo CUP.

3.5.1 Gramática para la calculadora.

Como se mencionó en secciones anteriores, las gramáticas libres de contexto tienen un símbolo terminal y otro no terminal. En este trabajo los terminales serían los tokens que recibimos, y los no terminales vamos a definirlos posteriormente en la gramática.

La gramática se declara con las siguientes reglas, donde los no terminales están declarados entre "<no terminal>":

<resultado> ::= <expresion>.
<expresion> ::= <expresion> "+" <termino> | <expresion> "-" <termino> | <termino>.
<termino> ::= <termino> "*" <menosUnario> | <termino> "/" <menosUnario> | <menosUnario>.
<menosUnario> ::= "-" <menosUnario> | <factor>
<factor> ::= <int> | <float> | "(" <expresion> ")" | <funcionAdicional>.
<funcionAdicional> ::= "ABS(" <expresion> ")" | "POTENCIA(" <expresion> "," <expresion> ")" | "RAIZ(" <expresion> "," <expresion> ")".

Con esta gramática podemos ver que hay por una parte suma y resta, y la otra donde hay multiplicación y división, donde en toda la gramática se respeta el principio de asociatividad y precedencia.

3.5.2 Archivo parser.cup

En la sección 3.5.1 se planteó la gramática libre de contexto que se utilizará para reconocer las funcionalidades de la calculadora, con esto en mente se prepara un archivo parser.cup donde declararemos dicha gramática aplicando la lista de tokens que se genera en el léxico.

Se declaran los paquete donde pertenece el archivo con la siguiente linea `package archivos.sintactico;`

Después debemos declarar las funcionalidades que en caso de que se produzca un error en el análisis, por eso en la parte de estructura de la seccion 3.3.2.1 pudimos ver que hay una sección llamada parser code, en ella agregaremos además del tratamiento de errores, una función que nos muestre las reglas que irá pasando el analizador sintáctico.

Para ello definimos primero la función concat_rules que tendrá como argumento un string. Además debemos inicializar una variable llamada reglas donde almacenaremos los string que represente las reglas en la GLC.

```
public void syntax_error(Symbol s) {
    concat_rules( regla: s.toString()+". Simbolo n° "+s.sym+ " no reconocido." );}

public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception{
        throw new Exception(s.toString()+"' . Simbolo n° "+s.sym+ "." ); }

public String reglas = "";

public void concat_rules(String regla) { reglas += regla + "\n"; }
```

Fig. 15. Parser Code.

Las líneas de código están entre `parser code {: código :}`

Siguiendo con la estructura del archivo, nos encontramos con la lista de símbolos, en ella declararemos los tokens que se generan en el léxico como terminales, y como no terminales los símbolos de la gramática.

```
terminal SUMA,COMA, RESTA, MULT, DIV, PARENTESISO, PARENTESISC;
terminal String ERROR,INT,FLOAT,POTENCIA,RAIZ,ABS;

nonterminal  Resultado resultado;
nonterminal Expresion expresion, factor, menos_unario, termino,funcionesAdicionales;
```

Fig. 15. Lista de Símbolos.

Podemos ver que en los no terminales, hay clases Resultado y Expresión que en las próximas secciones se explicará. También podemos observar que los terminales utilizados coinciden con los tokens que declaramos en el analizador léxico, esto es para que no se produzca ningún error.

A continuación se debe declarar en qué símbolo se tiene que empezar, en nuestro caso es el símbolo no terminal resultado.

```
start with resultado;
```

Ya declarados todos los pasos previos se pondrá parte del código donde se muestra la gramática y se llama a la función que concatena las reglas, además de la declaración de RESULT donde se instancian clases que se verán en la sección de AST.

```
49  resultado ::= expresion:e
50      {:  concat_rules("REGLA 0: resultado --> expresion" + "\n  --> " + e.darResultado());
51      RESULT = new  Resultado("Resultado",e);
52      :};
53
54  expresion ::=
55      expresion:e1 SUMA termino:e2 {:{...}:}
59      |
60      expresion:e1 RESTA termino:e2 {:{...}:}
64      |
65      termino:t {:{...}:};
69
70  termino ::=
71      termino:t MULT menos_unario:mu {:{...}:}
75      |
76      termino:t DIV menos_unario:mu {:{...}:}
80      |
81      menos_unario:mu {:{...}:};
```

Fig. 16. Reglas GLC 1.

```
86  menos_unario ::=
87      RESTA menos_unario:mu {:{...}:}
91      |
92      factor:f {:{...}:};
96
97  factor ::=
98      INT:i {:{...}:}
102     |
103     FLOAT:f {:{...}:}
107     |
108     PARENTESISO expresion:eo PARENTESISC {:{...}:}
112     |
113     funcionesAdicionales:fa{:{...}:}
117     ;
```

Fig. 17. Reglas GLC 2.

```
funcionesAdicionales::=
    ABS PARENTESISO expresion:e  PARENTESISC {:
            concat_rules("REGLA 5.1: funcionesAdicionales --> ABS ( expresion )  " + "\n  --> "+"ABS( "+e.darResultado()+")");
            RESULT = new ABS("ABS",e);
        :}
    |
    POTENCIA PARENTESISO expresion:e1 COMA expresion:e2 PARENTESISC {:
        concat_rules("REGLA 5.2: funcionesAdicionales --> POTENCIA ( expresion, expresion )  "
                    + "\n  --> "+"POTENCIA("+e1.darResultado()+","+e2.darResultado()+")");
        RESULT = new POTENCIA("Potencia",e1,e2);
    :}
    |
    RAIZ PARENTESISO expresion:e1 COMA expresion:e2 PARENTESISC {:
        concat_rules("REGLA 5.3: funcionesAdicionales --> POTENCIA ( expresion, expresion )  "
                + "\n  --> "+"POTENCIA("+e1.darResultado()+","+e2.darResultado()+")");
        RESULT = new RAIZ("Raiz",e1,e2);
        :}
    ;
```

Fig. 18. Reglas GLC 3.

Con estas reglas y las declaraciones, lo que nos queda es generar los archivos java correspondientes al léxico y al sintáctico, lo de la siguiente manera:

```
public class GenerarFlexyCup {
    /** @param args the command line arguments */
    public static void main(String[] args) {
        //si le cambio el path se rompe asique lo dejo así.
        String path = "C:\\Users\\santi\\Desktop\\FinalCiencias1\\CalculadoraCiencias1\\Calculadora\\src\\archivos\\lexico\\lexico.flex";
        generarLexer(path);
        String[] param = new String[5];
        param[0] = "-destdir";
        param[1] = "C:\\Users\\santi\\Desktop\\FinalCiencias1\\CalculadoraCiencias1\\Calculadora\\src\\archivos\\sintactico";
        param[2] = "-parser";
        param[3] = "MiParser";
        param[4] = "C:\\Users\\santi\\Desktop\\FinalCiencias1\\CalculadoraCiencias1\\Calculadora\\src\\archivos\\sintactico\\parser.cup";
        generarParser(param);
    }
```

Fig. 19. Código Generar Lexer y Parser.

```java
public static void generarLexer(String path){
    File file=new File(path);
    //...
    jflex.generator.LexGenerator generator = new jflex.generator.LexGenerator(file);
    generator.generate();
}
public static void generarParser(String[] param){
    try {
        Main.main(param);
    } catch (IOException ex) {
        Logger.getLogger(GenerarFlexyCup.class.getName()).log(Level.SEVERE,  msg: null, ex);
    } catch (Exception ex) {
        Logger.getLogger(GenerarFlexyCup.class.getName()).log(Level.SEVERE,  msg: null, ex);
    }
}
```

Fig. 20. Generar Lexer y Parser.

Una vez generado podemos usarlos desde nuestra interfaz llamando al lexer y parser.

Con lo siguiente podemos generar la lista de reglas junto con el recorrido que hacen.

```java
public void Parsing(String contenido) throws IOException, Exception {

    try {
        MiLexico lexer = new MiLexico(new FileReader(archivo));
        MiParser parser = new MiParser(lexer);
        parser.reglas = "";
        StringBuilder Reglas = new StringBuilder();
        StringBuilder Simbolos = new StringBuilder();
        Symbol p = parser.parse();
        Reglas.append(parser.reglas);
        txp3.setText(parser.reglas);
        txp3.setForeground(Color.BLACK);
        txp3.setFont(new Font( name: "Dialog", Font.BOLD, size: 18));
    }
    catch (Exception e){
        JOptionPane.showMessageDialog(rootPane, message: "Error : " + e.getMessage());
    }

}
```

Fig. 21. Función de Parsing.

Una vez realizada esta parte del código, podremos ver las reglas que recorre en el siguiente caso.

Entrada: 20*3

Lo que se muestra en pantalla, el recorrido de las reglas pero además como el proyecto se encuentra terminado se muestra el resultado de la operación.

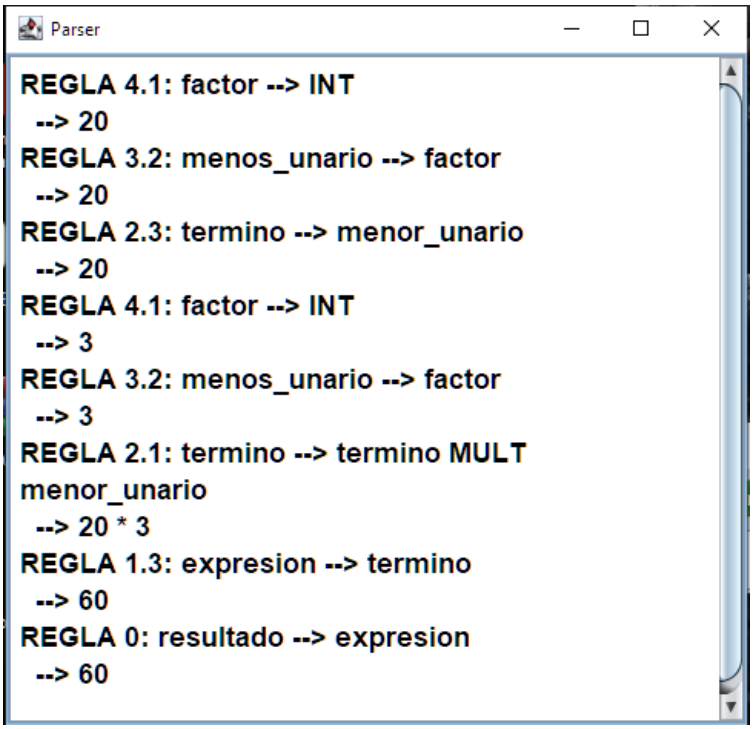

Fig. 22. Muestra de Parsing.

Analizando como se muestra y el recorrido de las reglas, utilizaremos JFLAP[11] para simularlo con esa aplicación.

En la siguiente imagen podemos observar la declaración de la GLC de tal forma que JFLAP pueda entenderla.

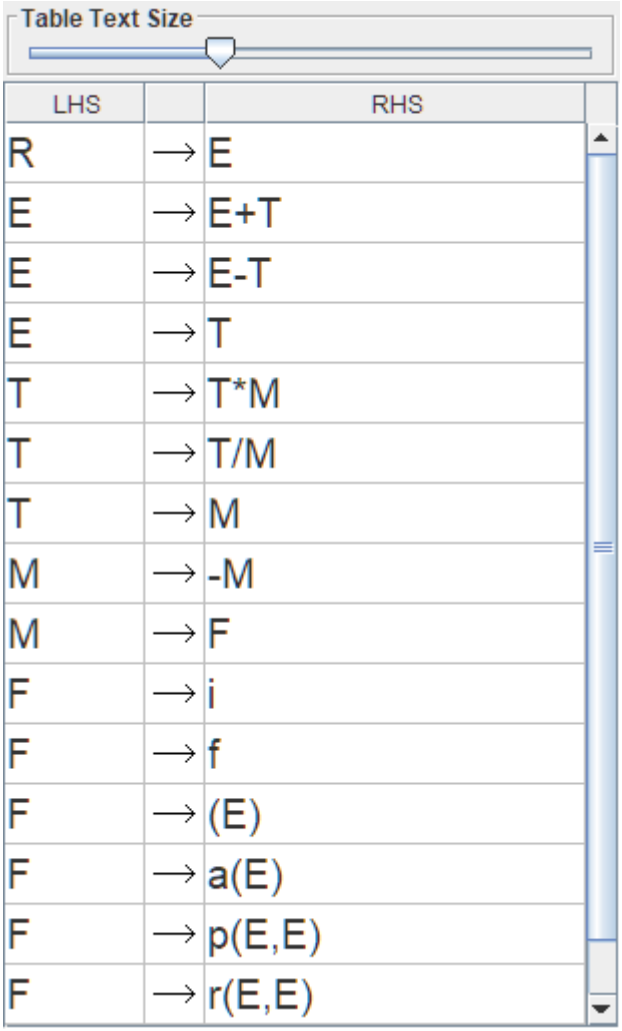

Fig. 23. GLC en JFLAP.

En JFLAP tenemos una opción para hacer una tabla de LR, dicha tabla nos sirve para saber que pasos seguir en cuanto se recibe una entrada, es una forma que nos ayuda a realizar el árbol de derivación que es lo que buscamos ver, o sea las reglas por lo que pasa.

Esta tabla de derivación también se genera en CUP, pero también podemos verla en varias páginas que la generan. En nuestro caso la siguiente tabla es para la gramática de la calculadora.

	(	)	*	+	,	-	/	a	f	i	p	r	$	E	F	M	R	T
0	s1					s2		s8	s9	s10	s11	s12		3	4	5	6	7
1	s1					s2		s8	s9	s10	s11	s12		13	4	5		7
2	s1					s2		s8	s9	s10	s11	s12			4	14		
3				s15		s16							r1					
4		r9	r9	r9	r9	r9	r9						r9					
5		r7	r7	r7	r7	r7	r7						r7					
6													acc					
7		r4	s17	r4	r4	r4	s18						r4					
8	s19																	
9		r11	r11	r11	r11	r11	r11						r11					
10		r10	r10	r10	r10	r10	r10						r10					
11	s20																	
12	s21																	
13		s22		s15		s16												
14		r8	r8	r8	r8	r8	r8						r8					
15	s1					s2		s8	s9	s10	s11	s12			4	5		23
16	s1					s2		s8	s9	s10	s11	s12			4	5		24
17	s1					s2		s8	s9	s10	s11	s12			4	25		
18	s1					s2		s8	s9	s10	s11	s12			4	26		
19	s1					s2		s8	s9	s10	s11	s12		27	4	5		7
20	s1					s2		s8	s9	s10	s11	s12		28	4	5		7
21	s1					s2		s8	s9	s10	s11	s12		29	4	5		7
22		r12	r12	r12	r12	r12	r12						r12					
23		r2	s17	r2	r2	r2	s18						r2					
24		r3	s17	r3	r3	r3	s18						r3					
25		r5	r5	r5	r5	r5	r5						r5					
26		r6	r6	r6	r6	r6	r6						r6					
27		s30		s15		s16												
28				s15	s31	s16												
29				s15	s32	s16												
30		r13	r13	r13	r13	r13	r13						r13					
31	s1					s2		s8	s9	s10	s11	s12		33	4	5		7
32	s1					s2		s8	s9	s10	s11	s12		34	4	5		7

Fig. 24. Tabla SLR.

En JFLAP podemos observar cómo se analizan entradas teniendo en la parte izquierda los estados, en la parte de arriba los símbolos. Dependiendo los símbolos se irá a un estado u otro.

Podemos dar el siguiente ejemplo.:

Entrada: i*i siendo i 20 y 3 respectivamente.

Generando el árbol de derivación en base a la entrada quedaría de igual en cuanto a la manera de recorrer las reglas con la Figura 22.

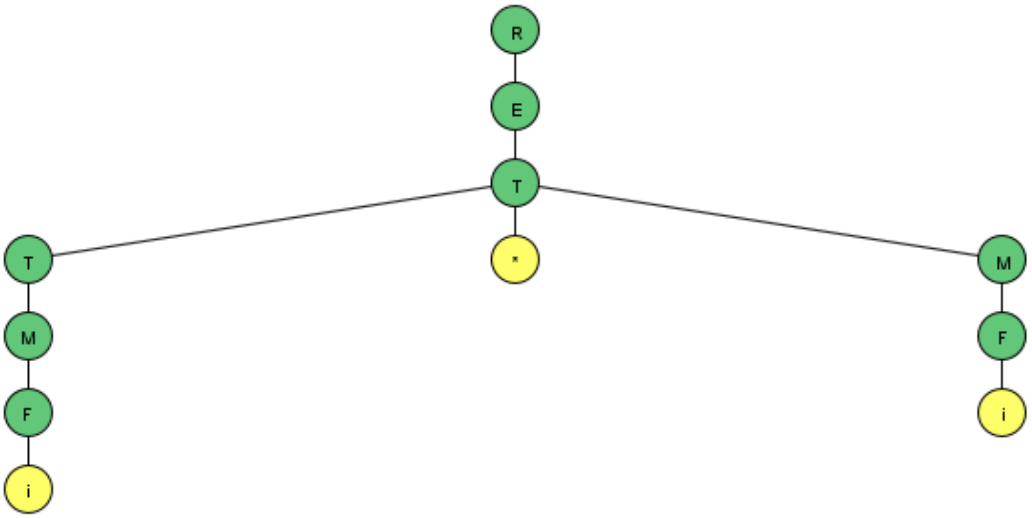

Fig. 24. Árbol de derivación.

Con esta comparativa podríamos probar tanto en nuestro programa como en JFLAP diferentes entradas y encontraríamos la misma salida en cuanto al recorrido.

En caso de que no respetemos esas reglas se mostrará un error como en la siguiente figura con entrada : 30(+3

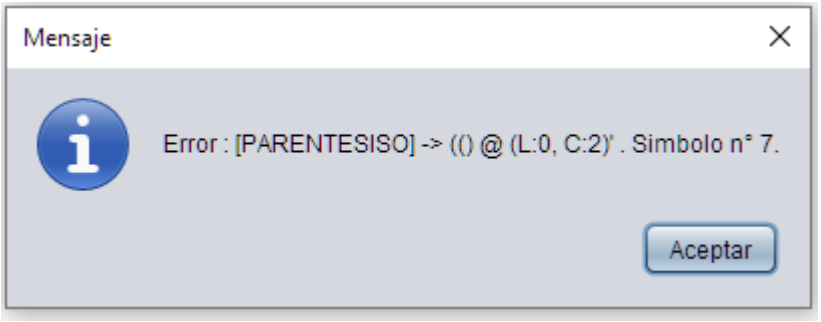

Fig. 25. Error sintáctico.

Podemos ver que nos dice el error en qué columna está, ya que nuestras reglas no contemplan el valor del símbolo '(' en ese lugar.

Cerrando esta sección podemos decir que el analizador sintáctico en el uso de nuestra calculadora nos ayuda a fijar las reglas que se deben seguir cuando recibimos una simple operación algebraica, y muestra el recorrido de dichas reglas.

3.6 AST

En lenguajes formales y lingüística computacional, un árbol de sintaxis abstracta (AST), o simplemente un árbol de sintaxis, es una representación de árbol de la estructura sintáctica simplificada del código fuente escrito en cierto lenguaje de programación[12], o en nuestro caso es la estructura sintáctica de una calculadora.

Podemos definirlo también como una forma de simplificar las reglas que se recorrieron, por eso podemos utilizarlo en la aplicación de la calculadora. Aquellos datos como delimitadores (paréntesis o comas) no se representarán.

Para poder hacer dicho árbol, primero debemos utilizar la librería Graphviz y posteriormente utilizar una estructura de clases que nos servirá para instanciar cada nodo del árbol.

3.6.1 Estructura de clases.

Como se vió en la sección 3.2 en el proyecto hay una carpeta con nombre ast, dicha carpeta tiene las clases que se utilizarán de nodos.

En la parte sintáctica se vió que hay clases Expresion y Resultado, esas clases se ven en la siguiente figura:

```
C:.
|
\---Base
    |   Nodo.java
    |   Resultado.java
    \---Expresiones
        |   Expresion.java
        +---Constantes
        |       Constante.java
        |       ConstanteEntera.java
        |       ConstanteFloat.java
        +---funcionesAdicionales
        |       ABS.java
        |       POTENCIA.java
        |       RAIZ.java
        \---Operaciones
            +---binarias
            |   |   OperacionBinaria.java
            |   \---arismeticos
            |           Division.java
            |           Multiplicacion.java
            |           Resta.java
            |           Suma.java
            \---unarias
                    MenosUnario.java
                    OperacionUnaria.java
```

Fig. 26. Estructura para AST.

Partiendo de la clase abstracta Nodo, la cual no solo tendrá la lógica de graficar el árbol sino también se agrega la función para dar el resultado de las expresiones, podremos arrancar con las demás clases.

```java
package archivos.ast.Base;

public abstract class Nodo{
    private String nombre;
    public Nodo() {}
    public Nodo(String nombre) { this.nombre = nombre; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public String getEtiqueta() {
        if (this.nombre != null) { return this.getNombre();}
        final String name = this.getClass().getName();
        final int pos = name.lastIndexOf( ch: '.') + 1;
        return name.substring(pos);
    }
    protected String getId() { return "nodo_" + this.hashCode(); }
    public String graficar(String idPadre){
        StringBuilder grafico = new StringBuilder();
        grafico.append(String.format("%1$s[label=\"%2$s\"]\n", this.getId(), this.getEtiqueta()));
        if(idPadre != null)
            grafico.append(String.format("%1$s--%2$s\n", idPadre, this.getId()));
        return grafico.toString();
    }
    public abstract Number darResultado();
```

Fig. 27. Implementación Clase Nodo.

La clase Nodo tiene como atributo un nombre, además de 3 funciones que serán comunes con todas las demás clases que extiendan de ella.
La función getEtiqueta() muestra el nombre y en algunas clases el valor que tienen o el resultado.
La función graficar() hará la funcionalidad de generar el árbol.
La función darResultado() da el resultado de la Expresion que tengan asociada, en caso de que corresponda a una operación aritmética o una función adicional tratará diferente la lógica de la función.
Podemos seguir las reglas de la GLC para instanciar las clases. Donde se observa que se instancia Resultado:

```
resultado ::= expresion:e
    {:  concat_rules("REGLA 0: resultado --> expresion" + "\n  --> " + e.darResultado());
    RESULT = new  Resultado("Resultado",e);
    :};
```

Fig. 26. Instancia Resultado.

La clase Resultado tiene la siguiente implementación

```
1   package archivos.ast.Base;
2   import archivos.ast.Base.Expresiones.Expresion;
3
4   public class Resultado extends Nodo{
5       public Expresion expresion;
6       public Resultado(String nombre, Expresion expresion) {super(nombre); this.expresion = expresion;}
7       public String getId() { return "nodo_resultado"; }
10      @Override
11      public String getEtiqueta() { return String.format("%s  \n %s", this.getNombre(), this.expresion.darResultado()); }
14      public String graficar() {
15          // Acá se dispara la invocación a los métodos graficar() de los nodos.
16          // Como no tiene padre, se inicia pasando null.
17          StringBuilder resultado = new StringBuilder();
18          resultado.append("graph G {");
19          resultado.append(this.graficar( idPadre: null));
20          resultado.append(this.expresion.graficar(this.getId()));
21          resultado.append("}");
22          return resultado.toString();
23      }
24      @Override
25      public Number darResultado() { return this.expresion.darResultado(); }
28  }
```

Fig. 27. Implementación clase Resultado.

Como podemos ver los constructores y también un atributo de tipo Expresion, que cuando instanciamos en la parte sintáctica lo ponemos como argumento.

Por parte de la clase Expresion tenemos las mismas funcionalidades ya que extiende de Nodo pero al ser una clase abstracta no las implementa.
Las clases que extienden de Expresión son las clases Constante, Operación Binaria y Operación Unaria.
Por parte de Constante tiene un atributo valor, dicho valor es el que obtenemos en los lexemas ya que ConstanteEntera y ConstanteFloat las instanciamos cuando detectamos un INT o un FLOAT respectivamente.

```
97   factor ::=
98       INT:i {:
99           concat_rules("REGLA 4.1: factor --> INT " + "\n  --> " + i);
100          RESULT = new ConstanteEntera(i,"ConstanteEntera");
101      :}
102      |
103      FLOAT:f {:
104          concat_rules("REGLA 4.2: factor --> FLOAT " + "\n  --> " + f);
105          RESULT = new ConstanteFloat(f,"ConstanteFloat");
106      :}
```

Fig. 28. Instancia ConstanteEntera y ConstanteFloat.

Cada clase va a dar su resultado, osea el valor del lexema y se graficara en la clase Constante.

Por parte de Operación Binaria podemos decir que tiene dos expresiones y un resultado, dichas expresiones también se instancian en las clases Suma, Resta, Multiplicación y División que extienden de Operación Binaria. A continuacion podemos ver como se instancia la clase Suma.

```
54   expresion ::=
55       expresion:e1 SUMA termino:e2 {:
56           concat_rules("REGLA 1.1: expresion --> expresion SUMA termino " + "\n  --> " + e1.darResultado() +" + "+ e2.darResultado());
57           RESULT = new Suma("Suma",e1,e2);
58       :}
```

Fig. 29. Regla donde se instancia clase Suma.

En las operaciones binarias hay que verificar los tipos de datos que tienen sus expresiones para saber si devolver una ConstanteEntera o una ConstanteFloat.

En la clase Operación unaria podemos decir que tiene una sola expresión, y dicha expresión en la funcion darResultado() se la devuelve en valor negativo.

Por último las funciones adicionales dependiendo de los argumentos que tenga la función podemos agregar n argumentos y con ello tendrá n expresiones. Dichas funciones extienden de Expresión, un ejemplo podemos poner a la función de potencia que tiene 2 argumentos.

```java
package archivos.ast.Base.Expresiones.funcionesAdicionales;
import archivos.ast.Base.Expresiones.Expresion;
public class POTENCIA extends Expresion {
    private Expresion arg1;
    private Expresion arg2;
    public POTENCIA(String nombre,Expresion a,Expresion b) {
        super(nombre); this.arg1=a; this.arg2=b;}
    @Override
    public String graficar(String idPadre) {
        StringBuilder resultado = new StringBuilder();
        resultado.append(super.graficar(idPadre));
        resultado.append(String.format("%1$s[label=\"%2$s\"]\n", this.getId()+1, "BASE"));
        resultado.append(String.format("%1$s--%2$s\n", this.getId(), this.getId()+1));
        resultado.append(this.arg1.graficar( idPadre: this.getId()+1));
        resultado.append(String.format("%1$s[label=\"%2$s\"]\n", this.getId()+2, "EXPONENTE"));
        resultado.append(String.format("%1$s--%2$s\n", this.getId(), this.getId()+2));
        resultado.append(this.arg2.graficar( idPadre: this.getId()+2));
        return resultado.toString();
    }
    @Override
    public Number darResultado() {
        return Math.pow(arg1.darResultado().doubleValue(),arg2.darResultado().doubleValue());
    }
}
```

Fig. 30. Implementación clase POTENCIA.

Podemos ver que darResultado() cambia con respecto a cada funcionalidad y también se ve con la función graficar() que cambia su implementación. Con esto dicho se pueden agregar muchas funciones a nuestra calculadora como por ejemplo sacar porcentaje o conversión de unidades.

Dicho todo esto podemos dar algunos ejemplos de entrada y saber el árbol que genera.

En los siguientes dos casos se mostrarán sus árboles.

Entrada 1 : 300*50-2+(abs(-400*3)*raiz(2,4))

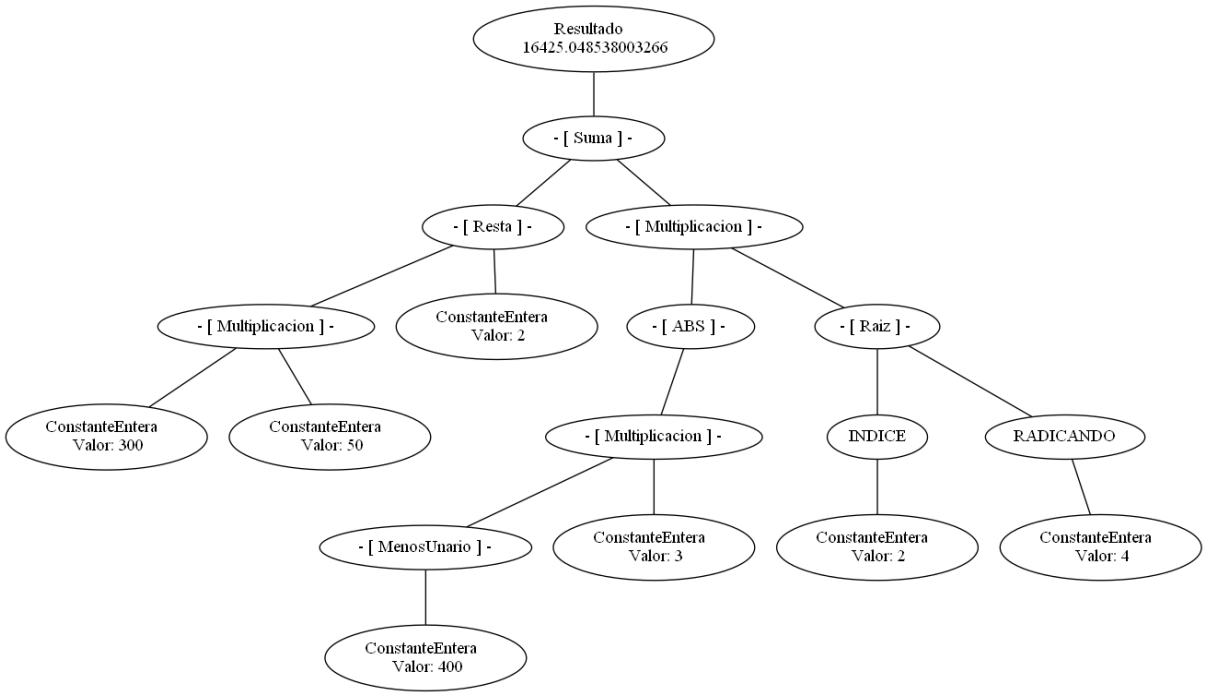

Fig. 31. AST Entrada 1.

Entrada 2: (-200*potencia(raiz(4,2),3))*-5

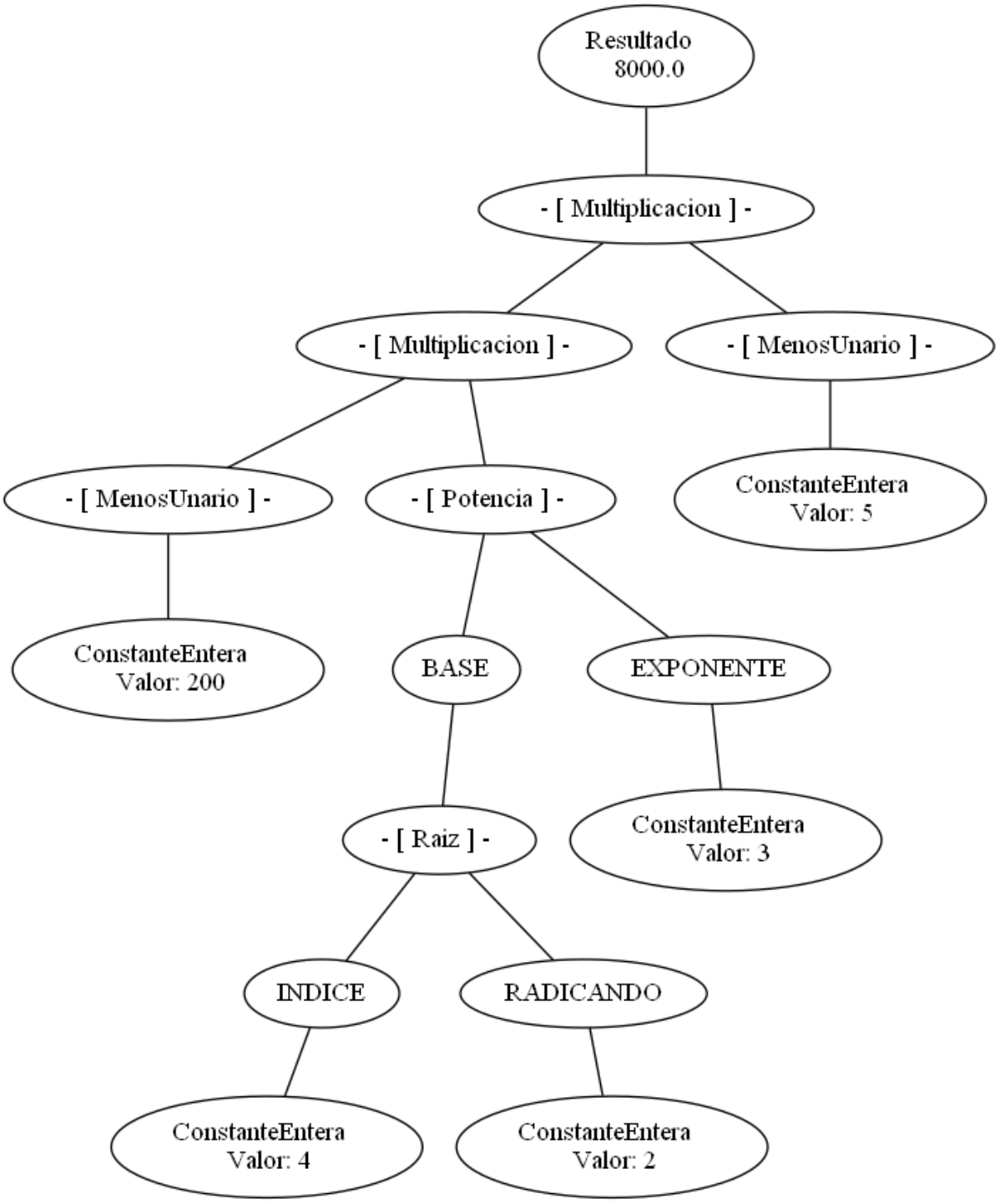

Fig. 32. AST Entrada 2.

Para finalizar la sección podemos decir que si bien no está dentro de los temas vistos en la materia es una buena forma de representar las operaciones que realiza la calculadora. Poder ver de forma simplificada si queremos agregar muchas operaciones juntas, en cambio en el parsing se mostrarían muchas reglas que nos dificultaría su seguimiento.

3.7 Interfaz de Usuario

Como parte del desarrollo de la calculadora podemos programar una sencilla interfaz que tome de entrada un texto, además para que quede constancia el trabajo pondremos varios botones que corresponden a las etapas por las que se pasaron en el desarrollo del proyecto.

Fig. 32. Interfaz de Calculadora.

Se ingresará una entrada e iremos por las etapas:
Entrada : 400 + 40 + 4 - 444

Fig. 33. Tokens reconocidos.

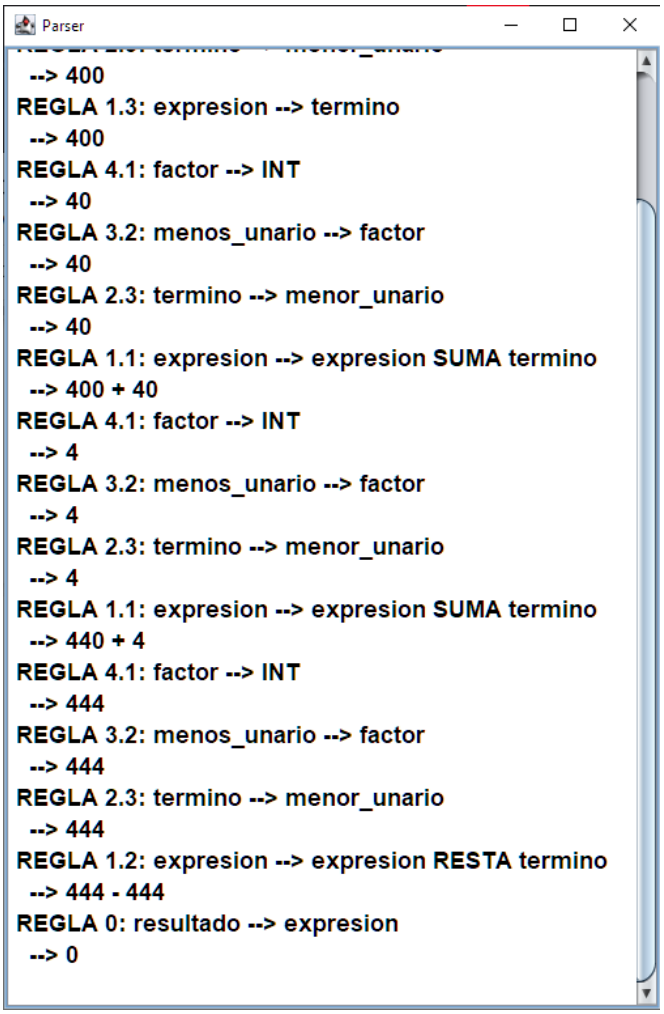

Fig. 34. Muestra parcial de Reglas.

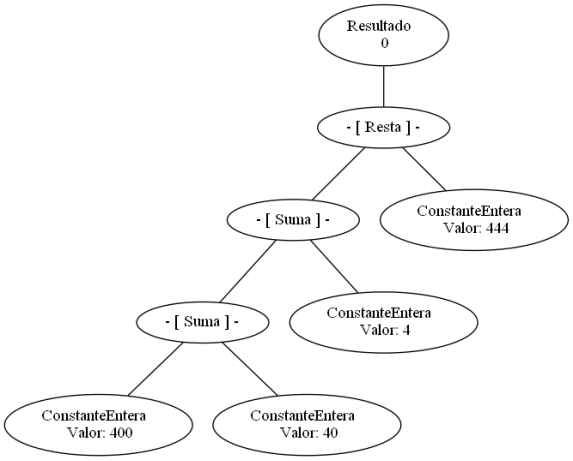

Fig. 35. Muestra AST.

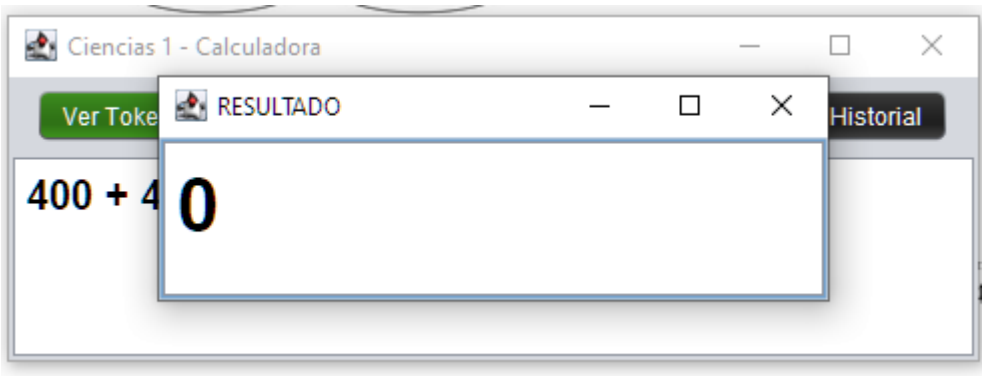

Fig. 36. Muestra resultado.

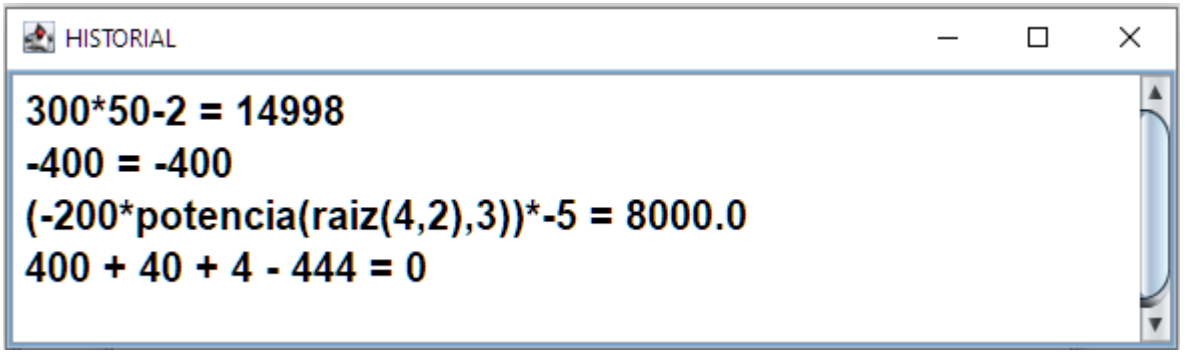

Fig. 37. Muestra Historial.

Adicionalmente se le agregó la funcionalidad de mostrar un historial que se encuentra presente en muchas calculadoras.

Para la elaboración de la interfaz se utilizó la librería Swing que viene presente Java, incluye widgets para interfaz gráfica de usuario tales como cajas de texto, botones, listas desplegables y tablas [13].

4 Conclusión

Como conclusión del trabajo se puede decir que la utilización de los conceptos adquiridos en la materia nos ayuda a comprender el funcionamiento interno y los pasos que realiza un analizador de una calculadora para validar las entradas, para darles acciones conforme a unas reglas sintácticas de una gramática y ver los pasos que lleva a cabo.

También es importante poder visualizar dichos pasos para que no se vuelvan una tarea difícil de seguir por eso el uso del AST para simplificar el recorrido.

A nivel personal puedo decir que fue una linda experiencia poder aplicar conocimientos adquiridos para realizar la aplicación, si bien el uso de la calculadora es limitado pude abordar la mayoría de los temas que vimos durante la cursada de la materia.

Como cierre de la conclusión, podemos decir que se cumplió el objetivo del trabajo. El cual fue desarrollar una aplicación con función de calculadora con ER y GLC, en base a un texto de entrada mostrar el resultado.