UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

DEPARTAMENTO DE COMPUTACIÓN Y SIMULACIÓN DE SISTEMAS

CARRERA LICENCIATURA EN INGENIERÍA DE SOFTWARE

# LENGUAJES FORMALES, AUTÓMATAS Y PROCESADORES DE LENGUAJES

## PROYECTO FINAL

FACILITADORA:

SHARON PÉREZ

ESTUDIANTES:

CASTILLO, ALFREDO (8-966-1822)

CHEN, DIANA (8-942-523)

KU, LUIS (8-923-509)

SALINAS, DALILA (8-934-2092)

TUÑÓN, DIEGO (8-938-468)

SEMESTRE II, 2020

# Introducción

Este proyecto de enfoque práctico tiene como objetivo el diseñar y programar un analizador léxico y sintáctico que evalué expresiones de cualquier tipo de archivo que emplea el lenguaje de programación **primoGEMA**. En primera instancia, se desarrollará la primera fase del compilador: el análisis léxico, donde para ello se definirá la tabla de símbolos que manejará la misma, pues de ésta dependerá la lectura de caracteres que se tienen como entradas. Como resultado se obtendrá la secuencia de componentes léxicos que empleará la segunda fase del compilador: el análisis sintáctico. En dicha fase, se comprueba que dichas sentencias son correctas para el lenguaje, enviando un error si no es el caso; a su vez, generan una representación, usualmente árboles, que corresponden a la sentencia que se esté analizando. Todo este proceso de desarrollo se verá evidenciado a continuación.

## Contenido

Un compilador es un programa constituido por varios subsistemas que trabajan en conjunto para traducir un texto, específicamente un lenguaje de programación de alto nivel, a código de máquina, a fin de que la computadora pueda entenderlo, procesarlo y ejecutarlo. El primer compilador fue desarrollado por Grace Hopper en 1952 con el objetivo de facilitarle a los programadores la escritura de aplicaciones.


*Ilustración 1 función del compilador*

Los diferentes subsistemas o fases de un compilador están constituidos en base a la clasificación de las gramáticas definidas por Noam Chomsky. La primera fase del compilador es el analizador léxico, el cual se encarga de procesar las oraciones de un lenguaje, identificar lexemas según las expresiones regulares definidas, y asignarle el token correspondiente a cada una. El analizador sintáctico, que hace referencia a la segunda fase del compilador, toma los tokens generados por el analizador léxico y determina si el orden en que se escribieron las palabras en la oración analizada es correcto o no. Posterior al análisis sintáctico, la oración analizada por el compilador pasa a través de un analizador semántico para determinar el sentido de la oración y en algunos casos, sigue con la generación de códigos intermedios y optimización de éste.

El presente trabajo pretende documentar el desarrollo las dos primeras fases de un compilador que reconoce el lenguaje de programación **primoGEMA**, el cual se caracteriza

principalmente porque los tipos de datos están definidos por los colores morado, azul, naranja, rojo y verde, y porque las palabras reservadas están definidas esencialmente en español, siguiendo una estructura semejante al pseudocódigo. También se caracteriza porque los identificadores inician obligatoriamente con la palabra "var_" y terminan con un número. A continuación, se detallarán los pasos seguidos para llevar a cabo el desarrollo de los analizadores.

## 1. Análisis léxico

Para la fase del analizador léxico del lenguaje **primoGEMA** se definió una tabla de símbolos que contiene los tokens con su correspondiente expresión regular y los lexemas que reconoce. En esencia, la tabla se clasificó en varias secciones en función de los tokens relacionados entre sí a fin de mantener una secuencia lógica y evitar confusiones. Las clasificaciones generales de la tabla incluyen: identificador, datos (dígitos, decimales, booleanos, letras, palabras), tipos (entero, booleano, cadena, carácter, doble), espacios y saltos, signos, operadores, aritmética, asignadores, condicionales, reservadas, apertura y cierre. En la tabla 1 se puede ver a detalle la especificación de cada token.

*Tabla 1 tabla de símbolos del lenguaje de programación primoGEMA.*

| Lexema | Expresión regular | Token |
|---|---|---|
| var_a1, var_b2, var_c3, … | /var_[a-zA-Z]+[0-9]+/ | Identificador |
| **Datos** | | |
| 1, 2, 3, 4, ... | /[0-9]/ | Dígitos |
| 1.2, 4.5, 6.22, … | /[0-9]+\.[0-9]+/ | Decimales |
| V, F | /V\|F/ | Booleanos |
| 'a', '9', '$' … | /\'.\'/ | Letras |

| | | |
|---|---|---|
| “abc1”, “9$a”, … | /\".+\"/ | Palabras |
| **Tipos** | | |
| MORADO | /MORADO/ | Entero |
| AZUL | /AZUL/ | Booleano |
| NARANJA | /NARANJA/ | Cadena |
| ROJO | /ROJO/ | Caracter |
| VERDE | /VERDE/ | Doble |
| **Espacios y saltos** | | |
| Vacio | /\s/ | Vacio |
| Salto | /\\[n r]/ | Salto |
| **Signos** | | |
| Final | /;/ | Final |
| **Operadores** | | |
| == | /==/ | IgualIgual |
| > | />/ | Mayor |
| >= | />=/ | Mayorigual |
| < | /</ | Menor |
| <= | /<=/ | Menorigual |
| != | /!=/ | Noigual |
| **Aritmética** | | |
| + | /+/ | Suma |
| - | /-/ | Resta |
| * | /*/ | Multiplicación |

| | | |
|---|---|---|
| / | /\// | División |
| % | /%/ | Porcentaje |
| ** | /**/ | Potencia |
| **Asignadores** | | |
| = | /=/ | Igual |
| += | /\+=/ | Sumavar |
| -= | /\-=/ | Restavar |
| *= | /*=/ | Multiplicacionvar |
| /= | /\/=/ | Divisionvar |
| %= | /%=/ | Porcentajevar |
| **= | /**=/ | Potenciavar |
| **Condicionales** | | |
| Mientras | /MIENTRAS/ | Mientras |
| Si | /SI/ | Si |
| **Reservadas** | | |
| Inicio | /INICIO/ | Inicio |
| Fin | /FIN/ | Fin |
| **Apertura** | | |
| { | /{/ | AperturaLlave |
| ( | /\(/ | AperturaParentesis |
| **Cierre** | | |
| } | /}/ | CierreLlave |
| ) | /\)/ | CierreParentesis |

### 1.1. Generador de análisis léxico

Una vez definida la tabla de símbolos, se optó por utilizar la librería Flex JS para implementar el analizador léxico a nivel funcional. La librería trabaja con el lenguaje de programación JavaScript. Permite establecer la configuración del analizador léxico, escribir expresiones regulares y definir las reglas. Cuando el analizador léxico se ejecuta, analiza las entradas en busca de cadenas que encajen en alguna de las expresiones regulares definidas, y en función de ello realiza una acción correspondiente a un código JavaScript.

## 2. Análisis sintáctico

Una vez el analizador léxico escanea el código para reconocer el patrón léxico establecido, genera tokens los cuales son enviados al analizador sintáctico, el cual revisa las reglas gramaticales que fueron establecidas. A continuación, se detalla la tupla G, que define la gramática establecida para el analizador sintáctico. Vn hace referencia al vocabulario no terminal, Vt al vocabulario terminal, S al axioma inicial y P a las reglas de producción o derivación.

$$G = (Vn, Vt, S, P)$$

***Vn*** = {PROG, BLOCK, INICIO_PROG, DECLARACION, MATOPERACION, INICIO_CONDICIONAL, FIN_CONDICIONAL, FIN_PROG}

***Vt*** = {dígitos, decimales, palabras, letras, booleanos, entero, booleano, doble, cadena, carácter, condicionales, operador, asignación, aritmética, identificador, inicio, fin, final, aperturaparentesis, cierreparentesis, aperturallave, cierrellave}

***S*** = PROG

***P*** = {

**BLOCK**→INICIO_PROG | DECLARACION | MATOPERACION | INICIO_CONDICIONAL | FIN_CONDICIONAL | FIN_PROG

**DECLARACION** → entero identificador asignacion digitos final |

doble identificador asignacion decimales final |

cadena identificador asignacion palabras final |

caracter identificador asignacion letras final |

booleano identificador asignacion booleanos final

**INICIO_PROG** → inicio aperturallave

**MATOPERACION**→ identificador asignacion (identificador | decimales | digitos ) (aritmetica ( identificador | decimales | digitos))* final

**INICIO_CONDICIONAL** → condicionales aperturaparentesis LOGOPERACION cierreparentesis aperturallave

**FIN_CONDICIONAL** → fin condicionales cierrellave

**LOGOPERACION** → (booleanos| identificador | digitos | decimales ) (operador (booleanos| identificador | digitos | decimales))*

**FIN_PROG** →fin cierrellave

}

## 2.1. Generador de análisis sintáctico

Una vez definida la gramática para el analizador sintáctico, se optó por implementarlo utilizando la librería Ohm. Ohm es un generador de analizador sintáctico que consta de una biblioteca y un lenguaje específico de dominio. Puede ser usado para varias cosas, sin embargo, en el presente caso se optará para crear rápidamente analizadores, intérpretes y compiladores de lenguajes de programación. Dicho lenguaje se basa en el análisis de gramáticas de expresión (PEG), que son una manera formal de describir la sintaxis, similar a las expresiones regulares y las gramáticas libres de contexto. Además, la biblioteca Ohm proporciona una interfaz de JavaScript (conocida como Ohm/JS), y separa las reglas gramaticales de las acciones semánticas, donde la gramática se concibe con la definición del idioma y la semántica determina qué hacer con las entradas que son consideradas válidas por dicho idioma. Otros beneficios que brinda Ohm son: mejor modularidad (la semántica y la gramática son independiente entre ellas), potabilidad, y eficiencia (no se evalúan acciones semánticas en caso de que las entradas de datos no sean válidas.).

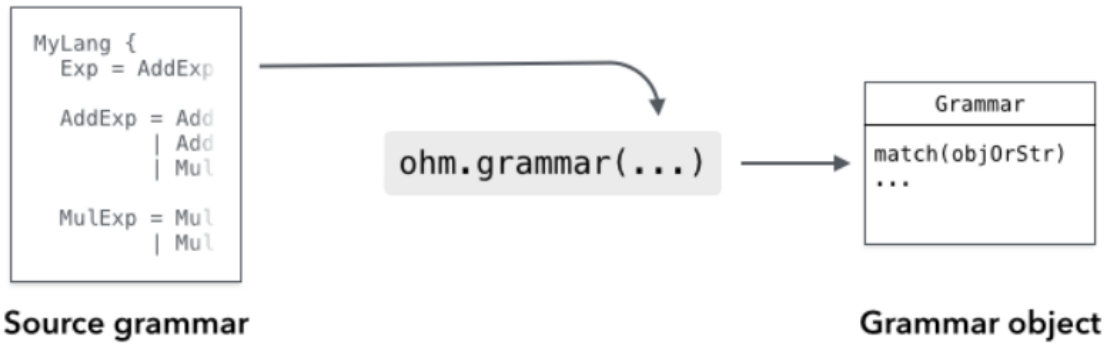



*Ilustración 2 proceso básico de Ohm.*

| Extra |
| --- |
| entero = "Morado" |
| booleano = "Azul" |
| doble = "Verde" |
| cadena = "Naranja" |
| caracter = "Rojo" |

*Ilustración 3 se denota el color para cada tipo de dato. Se verá reflejado a nivel de interfaz.*

## 3. Códigos de Prueba

Una vez implementado de manera funcional tanto el analizador léxico y sintáctico, se desarrollaron algunos ejemplos para comprobar el correcto funcionamiento de éstos en función del lenguaje definido.

### 3.1. Primera prueba

En la primera prueba se inicializó el programa con la palabra reservada INICIO seguida una llave de apertura. A continuación, se realizaron cinco declaraciones simples, una de cada tipo de dato definido. En la imagen se detalla el ejemplo según el árbol de derivación, seguido del código implementado y el resultado del análisis léxico y sintáctico.

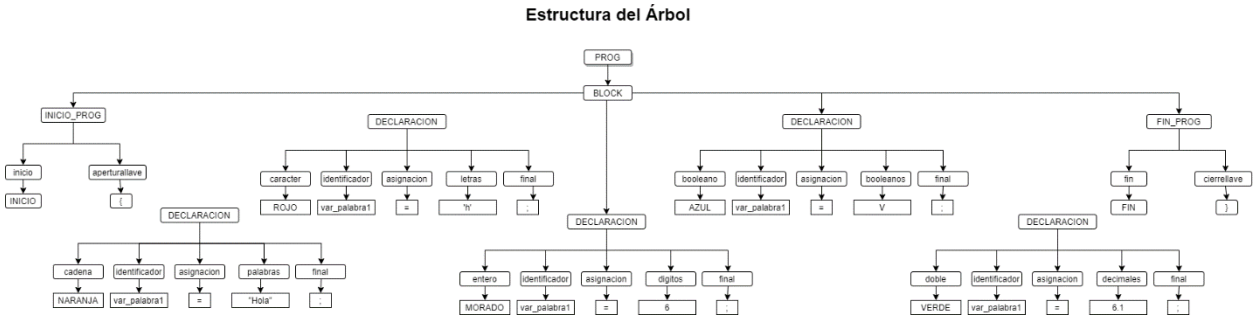



*Ilustración 4 estructura del árbol de la prueba 1.*

**Prueba 1: Análisis de todos los tipos de datos, asignadores y dígitos.**

```
INICIO {

NARANJA var_palabra1 = "Hola";

ROJO var_palabra1 = 'h';

MORADO var_palabra1 = 6;

AZUL var_palabra1 = V;

VERDE var_palabra1 = 6.1;


FIN }
```

*Ilustración 5 código de la prueba 1.*


*Ilustración 6 inserción del código de la prueba 1 a evaluar en la aplicación.*

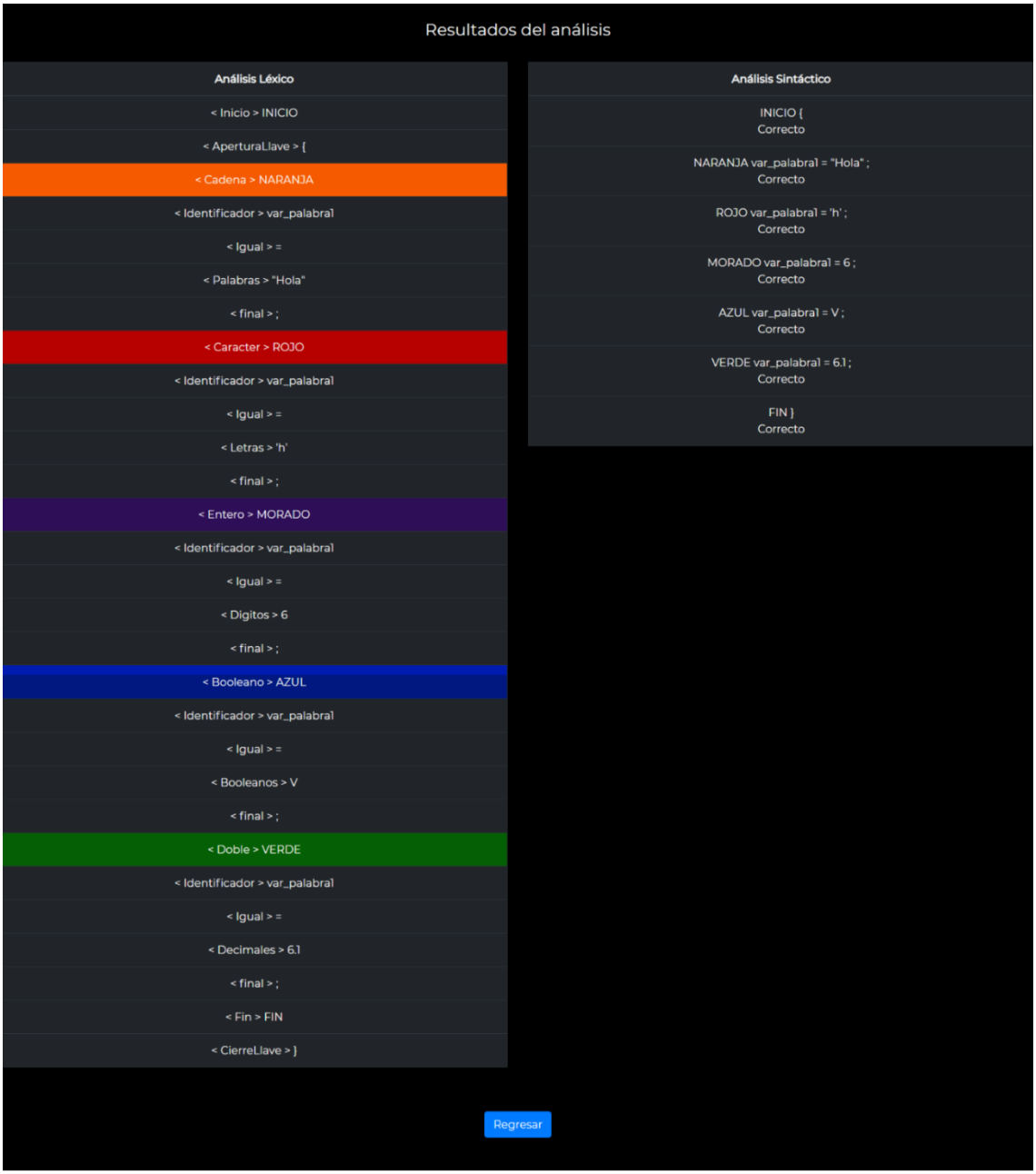



*Ilustración 7 resultado de la prueba 1 generado por parte del analizador léxico a la izquierda y del analizador sintáctico a la derecha.*

## 3.2. Segunda prueba

En la segunda prueba se inicia el programa con la palabra reservada INICIO seguida de la llave de apertura. Luego, se realiza unas operaciones, en donde, se resta

primero al identificador var_palabra1, variable de tipo doble, y luego a var_palabra2 se multiplica por 4. Finalmente, se compara si el digito 1 es mayor al digito 2 siendo este correcto y, por lo tanto, se realiza una declaración de un nuevo identificador, el var_palabra3, asignando el número 1.2. Termina la condicional y termina el programa. A continuación, se muestra el árbol de derivación seguido del código implementado y el resultado del análisis léxico y sintáctico.

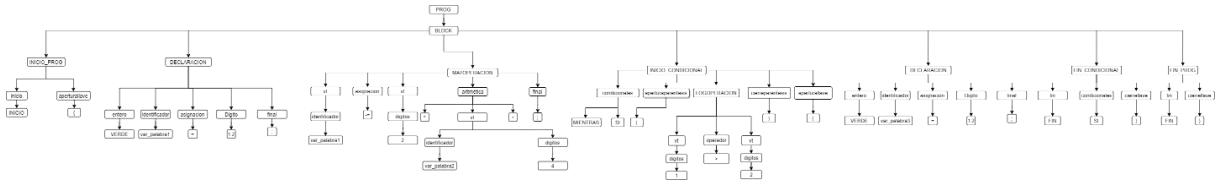


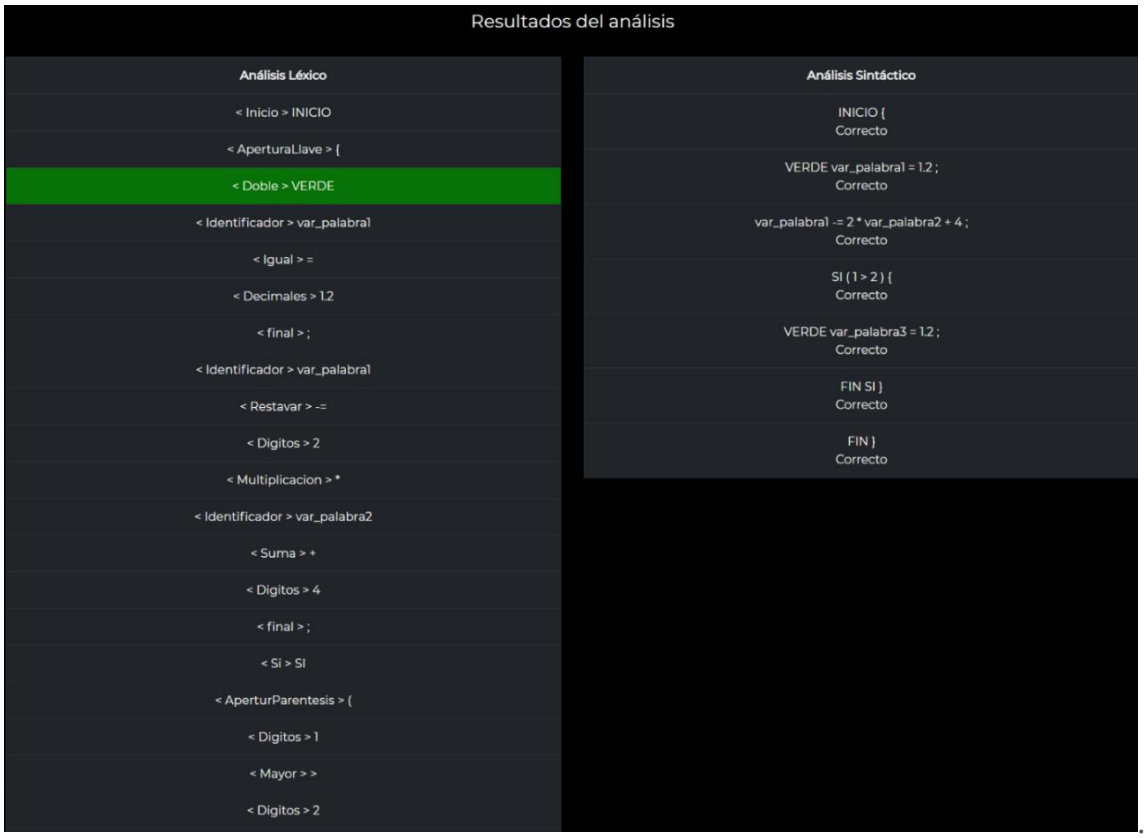
*Ilustración 8 estructura del árbol de la prueba 2.*

**Prueba 2: Análisis de operadores lógicos, aritméticos, asignadores.**

```
INICIO {
VERDE var_palabra1 = 1.2;
var_palabra1 -= 2 * var_palabra2 + 4;
SI(1>2){
VERDE var_palabra3 = 1.2;
FIN SI }
FIN }
```

*Ilustración 9 código de la prueba 2.*


*Ilustración 10 inserción del código de la prueba 2 a evaluar en la aplicación.*



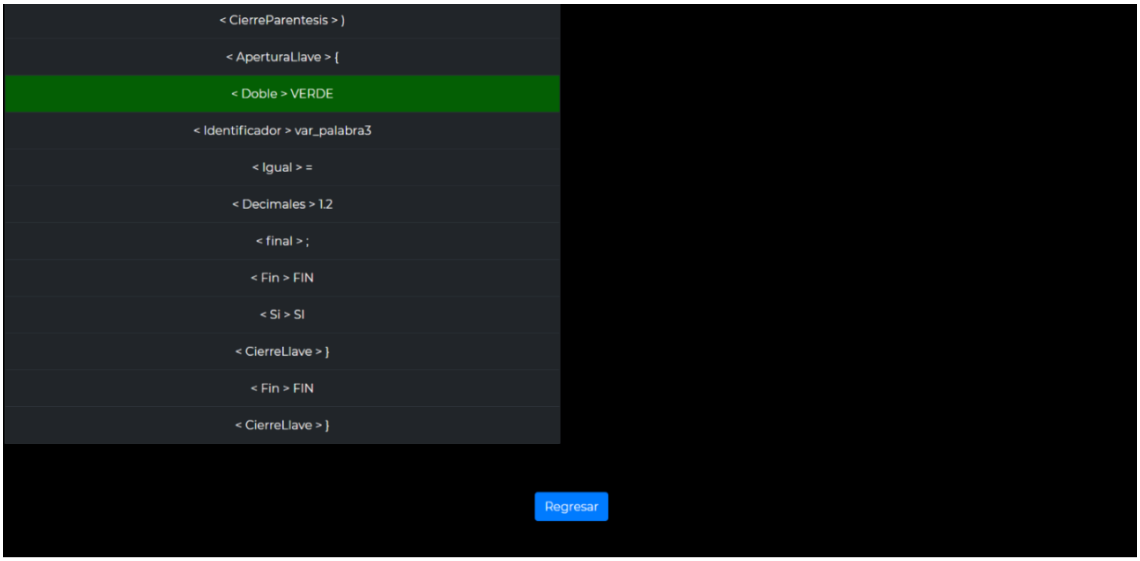



*Ilustración 11 resultado de la prueba 2 generado por parte del analizador léxico a la izquierda y del analizador sintáctico a la derecha.*

### 3.3. Tercera prueba

En la tercera prueba se inicia el programa con la palabra reservada INICIO seguida de la llave de apertura. Luego se realiza la declaración de una variable MORADA denotada por var_palabra2 la cual luego es reasignada a una operación matemática de multiplicación y suma en sí misma. Secuencialmente, se realiza una evaluación lógica (var_palabra2 >=1) dentro de un ciclo mientras el cual en caso de ser verdadera pasara por una segunda condicional donde se declara la variable VERDE var_palabra3 con 1.2, y una tercer condicional que potencia var_palabra2 a su potencia. A continuación, se muestra el árbol de derivación seguido del código implementado y el resultado del análisis léxico y sintáctico.

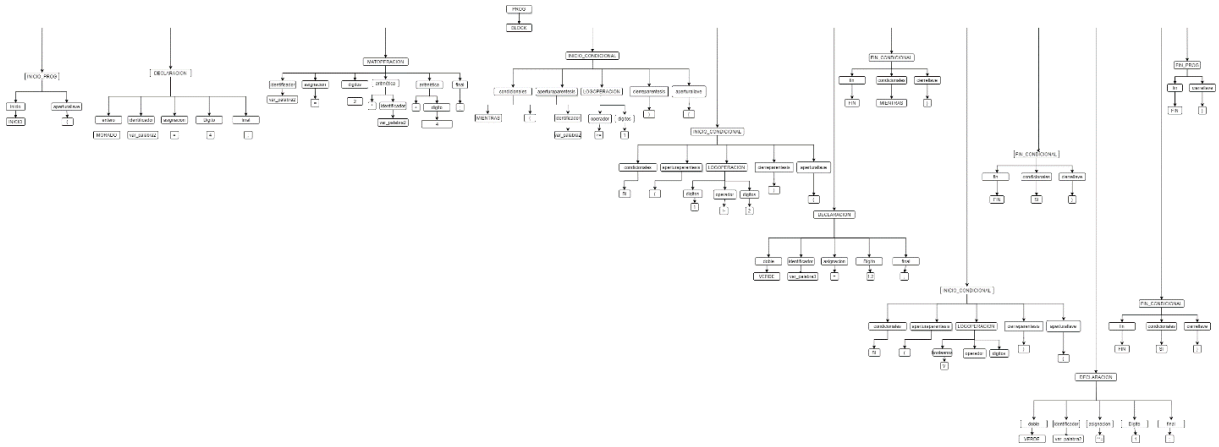


*Ilustración 12 estructura del árbol de la prueba 3.*

**Prueba 3: Análisis de operadores, datos, condicionales, palabras reservadas y signos.**

```
INICIO {

MORADO var_palabra2 = 4;

var_palabra1 = 2 * var_palabra2 + 4;

MIENTRAS(var_palabra2 >= 1) {

SI(1>2){

VERDE var_palabra3 = 1.2;

SI(V){

var_palabra2 **=1 ;

FIN SI}
```

```
FIN SI }

FIN MIENTRAS}

FIN }
```

*Ilustración 13 código de prueba 3.*

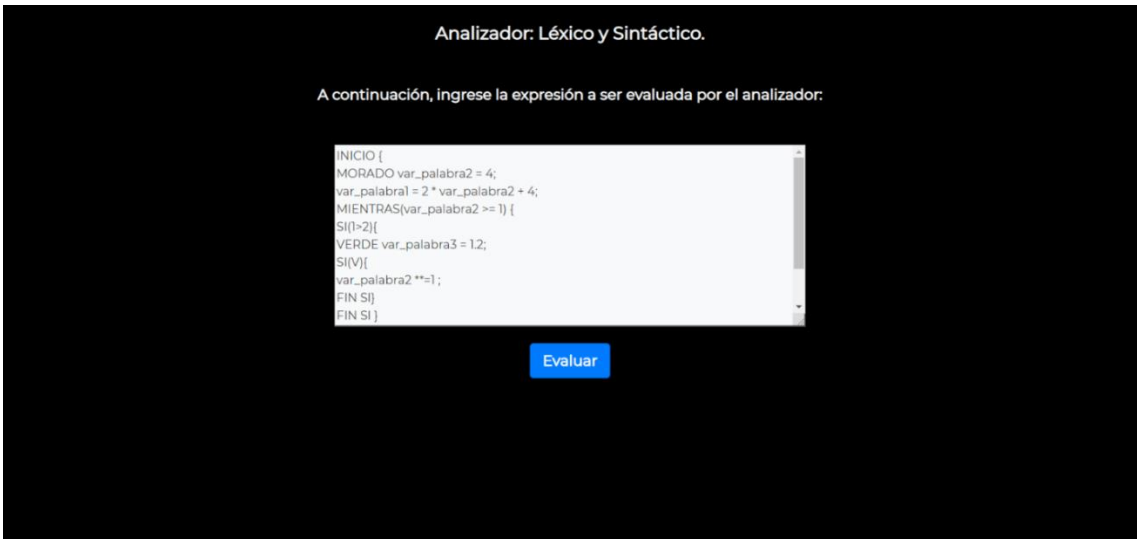



*Ilustración 14 inserción del código de la prueba 3 a evaluar en la aplicación.*

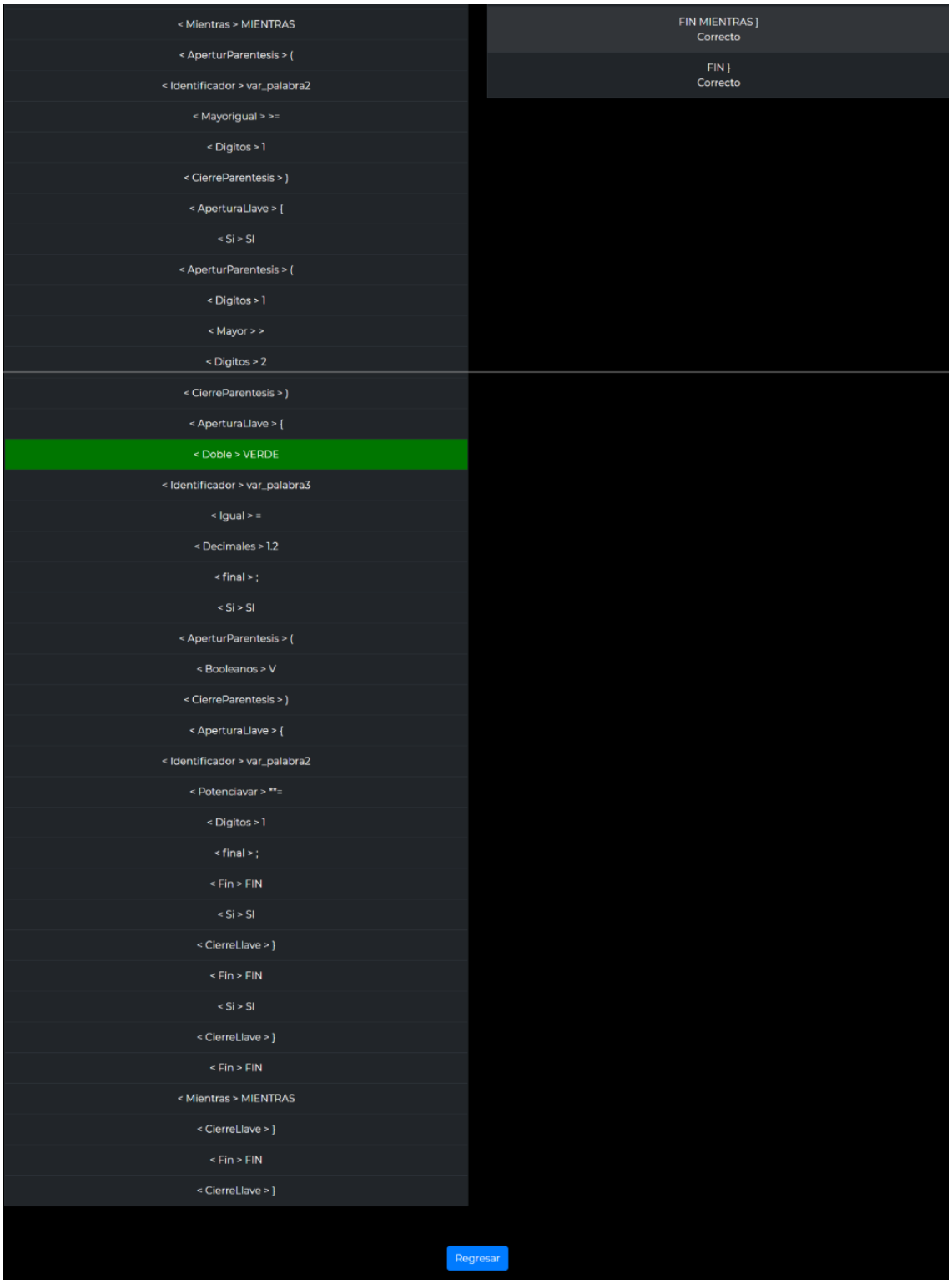



*Ilustración 15 resultado de la prueba 3 generado por parte del analizador léxico a la izquierda y del analizador sintáctico a la derecha.*

## 3.4. Cuarta prueba

En la cuarta prueba se inicia el programa con la palabra reservada INICIO seguida con una llave de apertura, después se presenta una declaración sobre el identificador var_palabra2 seguido de una operación al identificador var_palabra1 asignándole una operación combinado con dígitos, aritmética e identificadores. Luego, siguen tres condiciones respectivamente uno dentro del otro, la primera condición es un ciclo mientras con la lógica de un digito asignado a un identificador, después la segunda condición es una condición if donde se compara dos digito, dentro de ésta se hace una declaración al identificador var_palabra3. Al final la tercera condición es una condicional if donde se usa un booleano para esta que contiene una operación con el identificador var_palabra2. Finalmente se cierran las tres condiciones y se escribe la palabra reservada FIN. A continuación, se muestra el árbol de derivación seguido del código implementado y el resultado del análisis léxico y sintáctico.

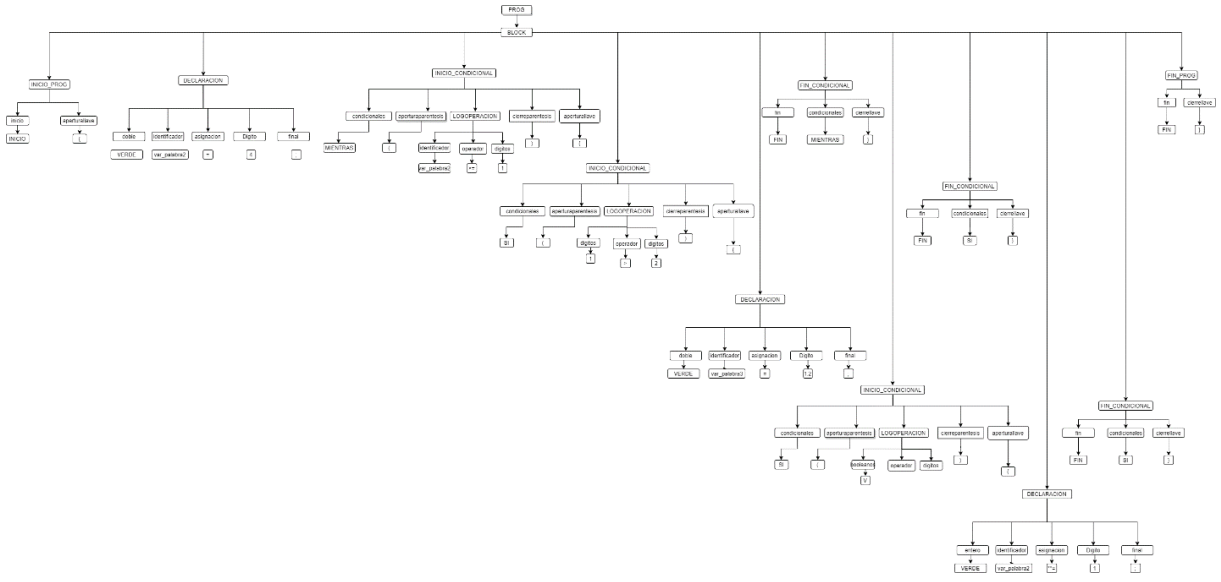


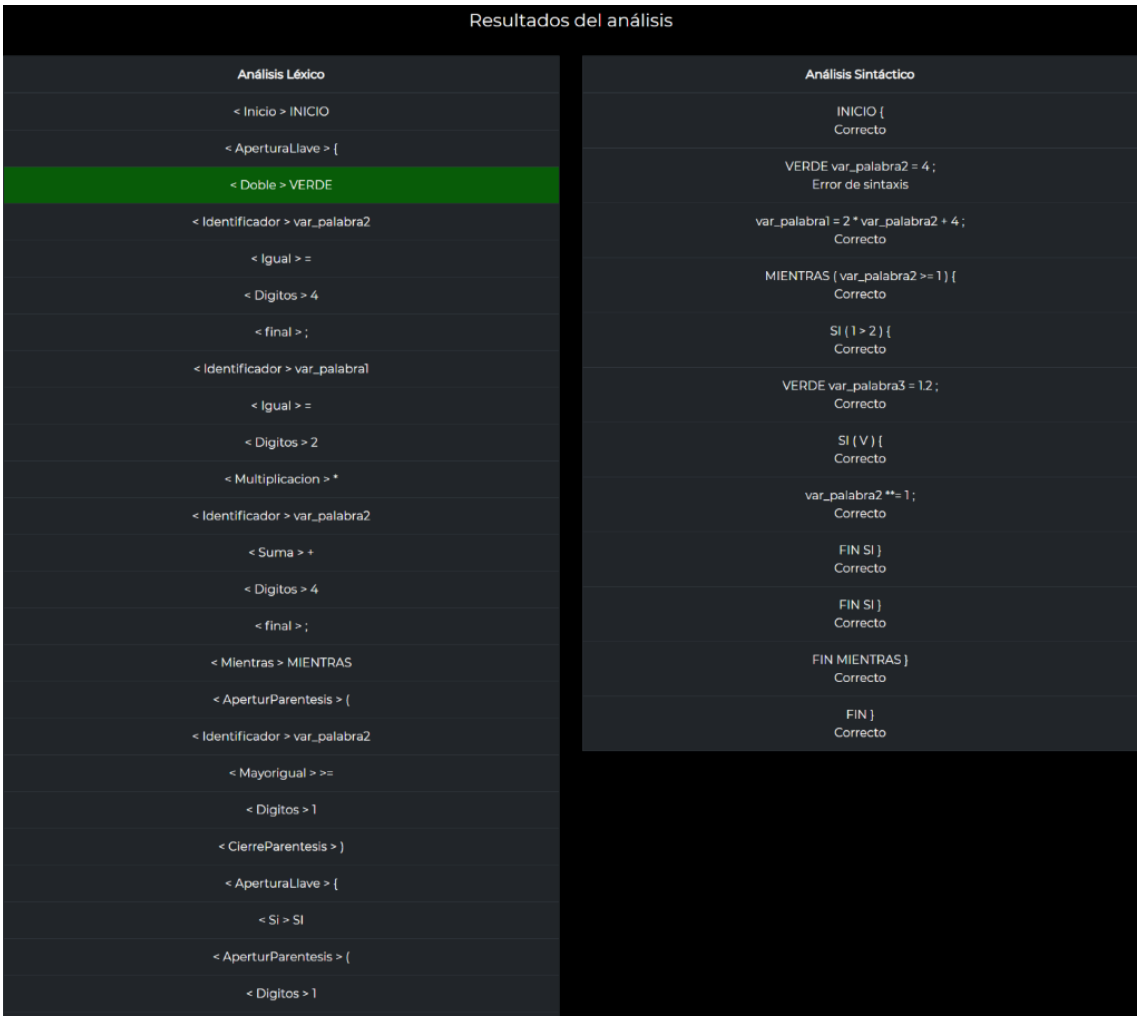
*Ilustración 16 estructura del árbol de la prueba 4.*

**Prueba 4: Error en el tipo de dato**

```
INICIO {

VERDE var_palabra2 = 4;

var_palabra1 = 2 * var_palabra2 + 4;

MIENTRAS(var_palabra2 >= 1) {

SI(1>2){

VERDE var_palabra3 = 1.2;

SI(V){

var_palabra2 **=1 ;

FIN SI}

FIN SI }

FIN MIENTRAS}

FIN }
```

*Ilustración 17 código de prueba 4.*


*Ilustración 18 inserción del código de la prueba 4 a evaluar en la aplicación.*



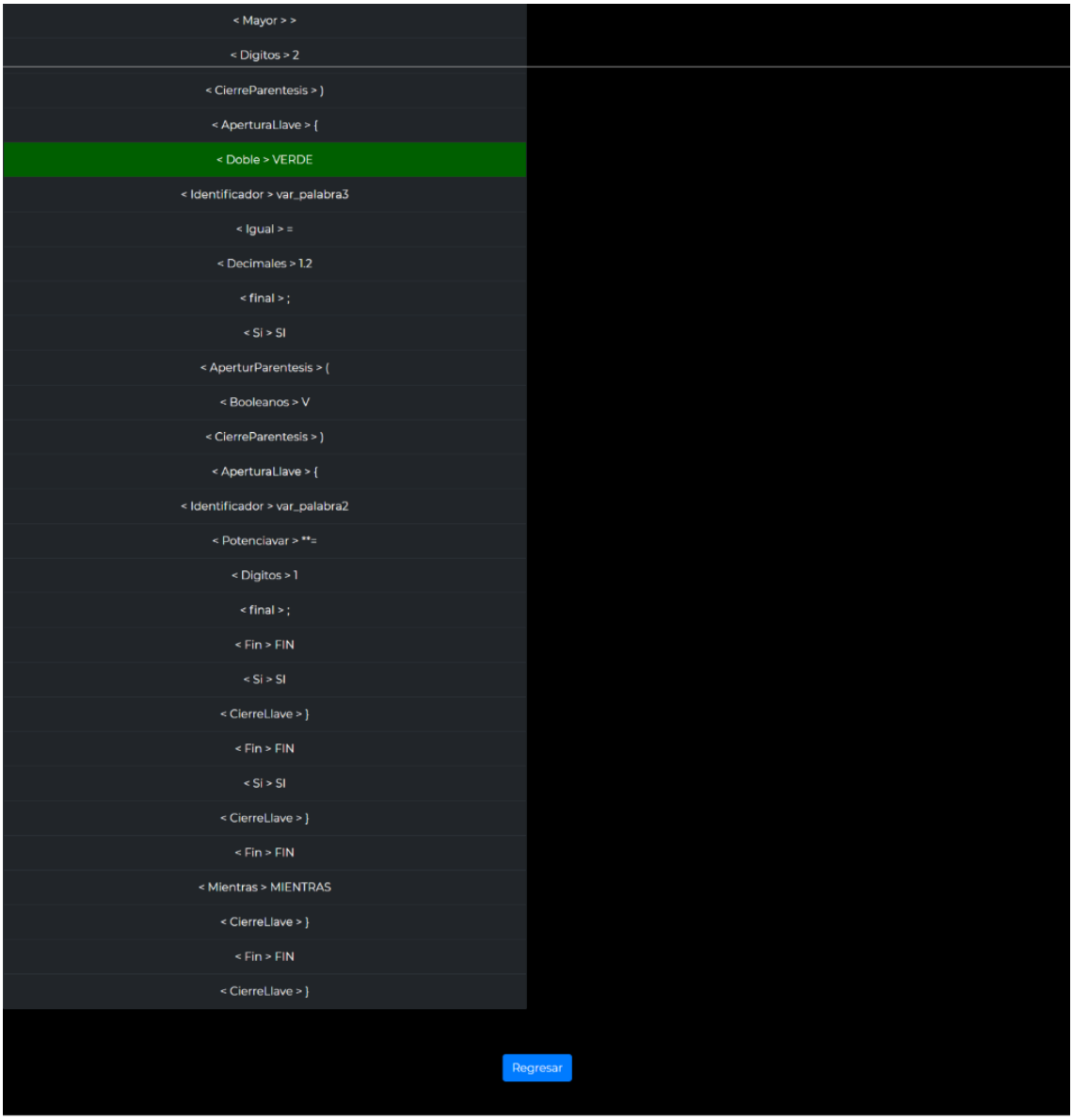



*Ilustración 19 resultado de la prueba 4 generado por parte del analizador léxico a la izquierda y del analizador sintáctico a la derecha.*

## 3.5. Quinta prueba

En la quinta prueba se inicia el programa con la palabra reservada INICIO, seguidamente se realiza una declaración donde var_palabra2, que es una variable de tipo doble, es igual a 4. Se procede a realizar una operación matemática donde la suma de 4 y

var_palabra2 multiplicado por 2 se le asigna a var_palabra1. Luego, se establece un ciclo en el cual mientras var_palabra2 sea mayor o igual a 1 se ejecutará el contenido de ésta. Su contenido consta de un if anidado; entrará al primero solo si 1 es mayor a 2. De ser así se realizará una declaración de tipo doble donde var_palabra3 es igual a 1.2. Seguido viene la segunda sentencia if a la que se entrará solo si lo anterior era verdadero, de ser así se ejecutara una operación matemática donde al valor que tiene var_palabra2 se le realizara una potencia a la 1. Finalmente, se cierran las dos sentencias if y se finaliza el ciclo mientras. Por último, se presenta la palabra reservada FIN. A continuación, se muestra el árbol de derivación seguido del código implementado y el resultado del análisis léxico y sintáctico.

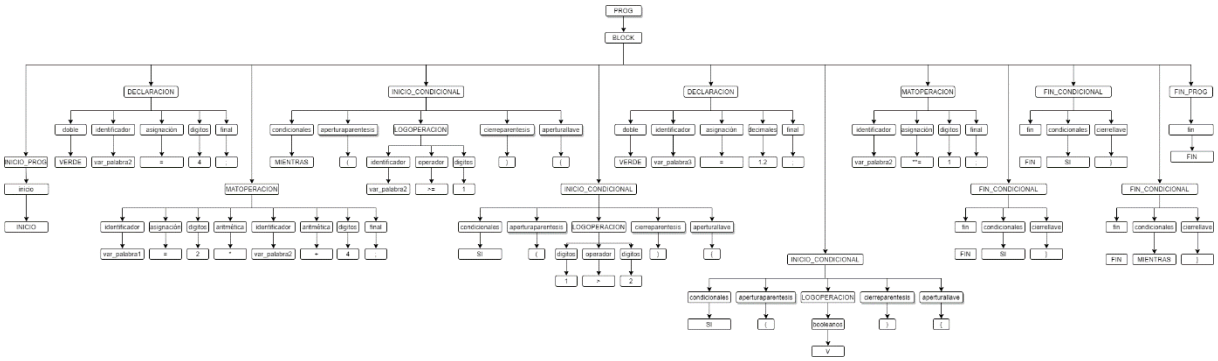



*Ilustración 20 estructura del árbol de la prueba 5.*

**Prueba 5: Error en el inicio y fin**

```
INICIO

VERDE var_palabra2 = 4;

var_palabra1 = 2 * var_palabra2 + 4;
```

```
MIENTRAS(var_palabra2 >= 1) {

SI(1>2){

VERDE var_palabra3 = 1.2;

SI(V){

var_palabra2 **=1 ;

FIN SI}

FIN SI }

FIN MIENTRAS}

FIN
```

*Ilustración 21 código de prueba 5.*


*Ilustración 22 inserción del código de la prueba 5 a evaluar en la aplicación.*


Resultados del análisis
Análisis Léxico
< Inicio > INICIO
< Doble > VERDE
< Identificador > var_palabra2
< Igual > =
< Digitos > 4
< final > ;
< Identificador > var_palabra1
< Igual > =
< Digitos > 2
< Multiplicacion > *
< Identificador > var_palabra2
< Suma > +
< Digitos > 4
< final > ;
< Mientras > MIENTRAS
< AperturParentesis > (
< Identificador > var_palabra2
< Mayorigual > >=
< Digitos > 1
< CierreParentesis > )
< AperturaLlave > {
Análisis Sintáctico
INICIO VERDE var_palabra2 = 4 ;
Error de sintaxis
var_palabra1 = 2 * var_palabra2 + 4 ;
Correcto
MIENTRAS ( var_palabra2 >= 1 ) {
Correcto
SI ( 1 > 2 ) {
Correcto
VERDE var_palabra3 = 1.2 ;
Correcto
SI ( V ) {
Correcto
var_palabra2 **= 1 ;
Correcto
FIN SI }
Correcto
FIN SI }
Correcto
FIN MIENTRAS }
Correcto
FIN FIN MIENTRAS }
Error de sintaxis
Error sintactico en el inicio
Error sintactico en el fin

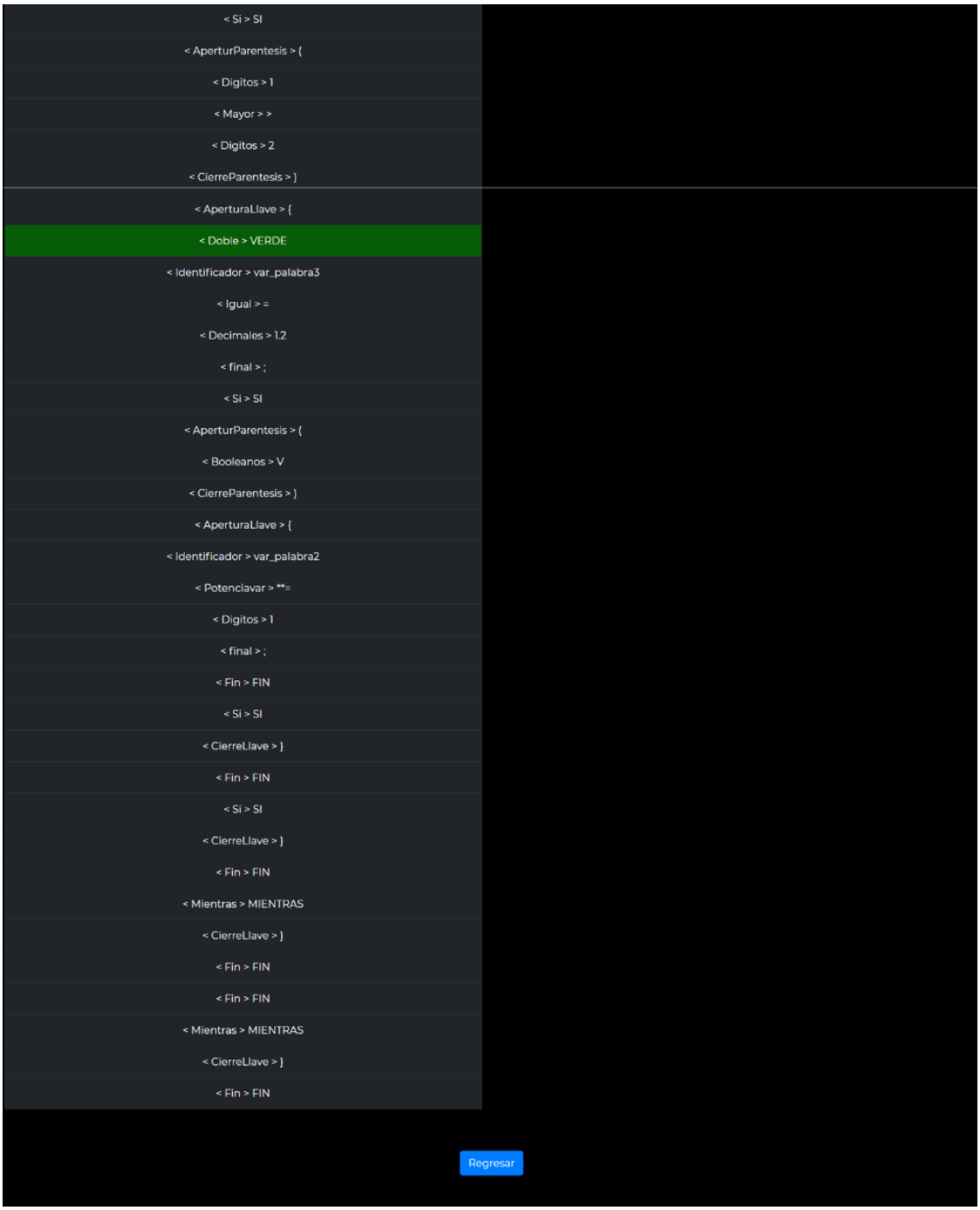



*Ilustración 23 resultado de la prueba 5 generado por parte del analizador léxico a la izquierda y del analizador sintáctico a la derecha.*

# Conclusiones

Los compiladores son herramientas que simplifican la complejidad del lenguaje de máquina, son un traductor, un intermediario entre el lenguaje entendible por las personas y el lenguaje entendible por las computadoras. Desarrollar un compilador no es una tarea fácil, pues requiere de un conocimiento profundo en diferentes áreas de la lingüística y un riguroso análisis lógico para definir la gramática y las derivaciones a fin de evitar la redundancia, facilitarle el trabajo al analizador semántico, mejorar el rendimiento y la detección de errores. Por ende, las tareas que realiza un compilador están compuesto por un conjunto de subcomponentes los cuales se encargan de realizar tareas específicas cuyos resultados son la entrada de otros. Visto esto, es importante reconocer el esfuerzo que conlleva el desarrollo de estas herramientas.

# Bibliografía

Charlead. (29 de Mayo de 2019). *JCup y JFlex | Analizador sintáctico con Java (explicación paso a paso)*. Obtenido de Youtube: https://www.youtube.com/watch?v=4Z6Tnit810Y&t=449s

Charlead. (17 de Febrero de 2019). *JFlex | Analizador léxico con Java (explicación paso a paso)*. Obtenido de Youtube: https://www.youtube.com/watch?v=5mIRrn2yEe8

Community, H. A. (2016). *Ohm*. Obtenido de GitHub: https://github.com/harc/ohm

Long, J. (2016). *Ohm*. Obtenido de GitHub: https://ohmlang.github.io/

*Ohm Examples*. (s.f.). Obtenido de Loyola Marymount University: https://cs.lmu.edu/~ray/notes/ohmexamples/

Sormy. (2 de Abril de 2017). *FLEX.JS*. Obtenido de GitHub: https://github.com/sormy/flex-js

Wikipedia, C. d. (6 de Noviembre de 2020). *Historia de la construcción de los compiladores*. Obtenido de Wikipedia La Enciclopedia Libre: https://es.wikipedia.org/w/index.php?title=Especial:Citar&page=Historia_de_la_construcci%C3%B3n_de_los_compiladores&id=130700497&wpFormIdentifier=titleform