



UNIVERSIDAD DE CÓRDOBA  
ESCUELA POLITÉCNICA SUPERIOR DE  
CÓRDOBA

INGENIERÍA INFORMÁTICA  
MENCIÓN EN COMPUTACIÓN  
3º CURSO - 2º CUATRIMESTRE

PROCESADORES DE LENGUAJES

**Intérprete de pseudocódigo en  
español: interpreter**

*Valentín Gabriel Avram Aenachioei  
Víctor Rojas Muñoz*

Año académico 2022-2023  
Córdoba, 6 de Junio, 2023

# Índice

|   |    |
|---|----|
| Índice de tablas  | IV |
| Índice de figuras   | VI |
| 1. Introducción   | 1  |
| 2. Lenguaje de Pseudocódigo                                   | 2  |
| 2.1. Componentes léxicos . . . . .                            | 2  |
| 2.1.1. Palabras reservadas . . . . .                          | 2  |
| 2.1.2. Identificadores . . . . .                              | 3  |
| 2.1.3. Números . . . . .                                      | 3  |
| 2.1.4. Cadena . . . . .                                       | 4  |
| 2.1.5. Operador de asignación . . . . .                       | 4  |
| 2.1.6. Operadores aritméticos . . . . .                       | 4  |
| 2.1.7. Operador alfanumérico . . . . .                        | 5  |
| 2.1.8. Operadores relacionales de números y cadenas . . . . . | 5  |
| 2.1.9. Operadores lógicos . . . . .                           | 6  |
| 2.1.10. Comentarios . . . . .                                 | 6  |
| 2.1.11. Punto y coma . . . . .                                | 6  |
| 2.2. Sentencias . . . . .                                     | 7  |
| 2.2.1. Asignación . . . . .                                   | 7  |
| 2.2.2. Lectura . . . . .                                      | 7  |
| 2.2.3. Escritura . . . . .                                    | 8  |
| 2.2.4. Comandos especiales . . . . .                          | 8  |
| 2.3. Sentencias de control . . . . .                          | 8  |
| 2.3.1. Sentencia condicional simple . . . . .                 | 9  |
| 2.3.2. Sentencia condicional compuesta . . . . .              | 9  |
| 2.3.3. Bucle while . . . . .                                  | 9  |
| 2.3.4. Bucle repeat . . . . .                                 | 9  |
| 2.3.5. Bucle for . . . . .                                    | 10 |
| 2.3.6. Estructura case . . . . .                              | 10 |
| 3. Tabla de símbolos  | 11 |
| 4. Análisis Léxico  | 16 |
| 4.1. Zona de declaraciones . . . . .                          | 16 |
| 4.2. Reglas normales . . . . .                                | 17 |

|   |           |
|---|-----------|
| <b>5. Análisis Sintáctico</b>                       | <b>26</b> |
| 5.1. Símbolos terminales . . . . .                  | 26        |
| 5.2. Símbolos no terminales . . . . .               | 28        |
| 5.3. Reglas de producción de la gramática . . . . . | 29        |
| 5.4. Acciones semánticas . . . . .                  | 31        |
| 5.4.1. program . . . . .                            | 31        |
| 5.4.2. stmtlist . . . . .                           | 32        |
| 5.4.3. stmt . . . . .                               | 32        |
| 5.4.4. type_of . . . . .                            | 34        |
| 5.4.5. block . . . . .                              | 35        |
| 5.4.6. controlSymbol . . . . .                      | 35        |
| 5.4.7. if . . . . .                                 | 36        |
| 5.4.8. while . . . . .                              | 36        |
| 5.4.9. for . . . . .                                | 37        |
| 5.4.10. case . . . . .                              | 38        |
| 5.4.11. valueList . . . . .                         | 38        |
| 5.4.12. value . . . . .                             | 39        |
| 5.4.13. cond . . . . .                              | 39        |
| 5.4.14. asgn . . . . .                              | 40        |
| 5.4.15. clear_screen . . . . .                      | 41        |
| 5.4.16. place . . . . .                             | 41        |
| 5.4.17. print . . . . .                             | 42        |
| 5.4.18. print_string . . . . .                      | 42        |
| 5.4.19. read . . . . .                              | 43        |
| 5.4.20. read_string . . . . .                       | 43        |
| 5.4.21. repeat . . . . .                            | 44        |
| 5.4.22. exp . . . . .                               | 44        |
| 5.4.23. listOfExp . . . . .                         | 50        |
| 5.4.24. restListOfExp . . . . .                     | 51        |
| <b>6. Código de AST</b>                             | <b>52</b> |
| 6.1. Clase Statement . . . . .                      | 52        |
| 6.1.1. AssignmentStmt . . . . .                     | 54        |
| 6.1.2. BlockCaseValueNode . . . . .                 | 54        |
| 6.1.3. ClearScreenStmt . . . . .                    | 54        |
| 6.1.4. EmptyStmt . . . . .                          | 54        |
| 6.1.5. ForStmt . . . . .                            | 54        |
| 6.1.6. IfStmt . . . . .                             | 55        |
| 6.1.7. PlaceStmt . . . . .                          | 55        |
| 6.1.8. PrintStmt . . . . .                          | 55        |
| 6.1.9. PrintStringStmt . . . . .                    | 55        |

|  |           |
|--|-----------|
| 6.1.10. ReadStmt . . . . .                 | 55        |
| 6.1.11. ReadStringStmt . . . . .           | 55        |
| 6.1.12. RepeatStmt . . . . .               | 56        |
| 6.1.13. TypeOfStmt . . . . .               | 56        |
| 6.1.14. WhileStmt . . . . .                | 56        |
| 6.1.15. BlockStmt . . . . .                | 56        |
| 6.2. Clase ExpNode . . . . .               | 57        |
| 6.2.1. StringNode . . . . .                | 60        |
| 6.2.2. ConcatenationNode . . . . .         | 60        |
| 6.2.3. IntegerDivisionNode . . . . .       | 60        |
| <b>7. Funciones auxiliares</b>             | <b>60</b> |
| <b>8. Modo de obtención del intérprete</b> | <b>61</b> |
| 8.1. Directorio /ast . . . . .             | 61        |
| 8.2. Directorio /error . . . . .           | 62        |
| 8.3. Directorio /includes . . . . .        | 62        |
| 8.4. Directorio /parser . . . . .          | 62        |
| 8.5. Directorio /table . . . . .           | 62        |
| 8.6. Directorio /examples . . . . .        | 62        |
| 8.7. Directorio /html . . . . .            | 63        |
| 8.8. Directorio raíz . . . . .             | 63        |
| <b>9. Modo de ejecución del intérprete</b> | <b>64</b> |
| 9.1. Modo interactivo . . . . .            | 64        |
| 9.2. Ejecución desde un fichero . . . . .  | 64        |
| <b>10. Ejemplos</b>                        | <b>65</b> |
| 10.1. Ejemplos precreados . . . . .        | 65        |
| 10.1.1. binario.p . . . . .                | 65        |
| 10.1.2. conversion.p . . . . .             | 66        |
| 10.1.3. menu.p . . . . .                   | 67        |
| 10.2. Ejemplos propios . . . . .           | 73        |
| 10.3. fibonacci.p . . . . .                | 73        |
| 10.4. geometrico.p . . . . .               | 77        |
| <b>11. Conclusiones</b>                    | <b>82</b> |
| <b>12. Bibliografía</b>                    | <b>83</b> |

## **Índice de tablas**

|    |   |    |
|----|---|----|
| 1. | Componentes léxicos funcionales . . . . . | 2  |
| 2. | Tabla de Símbolos . . . . .               | 11 |

# Índice de figuras

|     |  |    |
|-----|--|----|
| 1.  | Jerarquía de las clases de la tabla símbolos . . . . . | 13 |
| 2.  | Declaración de la clase StringVariable . . . . .       | 15 |
| 3.  | Declaraciones básicas del análisis léxico . . . . .    | 16 |
| 4.  | Estados usados en el análisis léxico . . . . .         | 17 |
| 5.  | Análisis léxico. Reglas I . . . . .                    | 17 |
| 6.  | Análisis léxico. Reglas II . . . . .                   | 18 |
| 7.  | Análisis léxico. Reglas III . . . . .                  | 18 |
| 8.  | Análisis léxico. Reglas IV . . . . .                   | 19 |
| 9.  | Análisis léxico. Reglas V . . . . .                    | 20 |
| 10. | Análisis léxico. Reglas VI . . . . .                   | 21 |
| 11. | Análisis léxico. Reglas VII . . . . .                  | 22 |
| 12. | Análisis léxico. Reglas VIII . . . . .                 | 22 |
| 13. | Análisis léxico. Reglas IX . . . . .                   | 23 |
| 14. | Análisis léxico. Reglas X . . . . .                    | 23 |
| 15. | Análisis léxico. Reglas XI . . . . .                   | 24 |
| 16. | Análisis léxico. Reglas XII . . . . .                  | 25 |
| 17. | Análisis léxico. Reglas XIII . . . . .                 | 25 |
| 18. | Acción semántica símbolo program . . . . .             | 31 |
| 19. | Acción semántica símbolo stmtlist . . . . .            | 32 |
| 20. | Acción semántica símbolo stmt I . . . . .              | 33 |
| 21. | Acción semántica símbolo stmt II . . . . .             | 34 |
| 22. | Acción semántica símbolo type_of . . . . .             | 34 |
| 23. | Acción semántica símbolo block . . . . .               | 35 |
| 24. | Acción semántica símbolo controlSymbol . . . . .       | 35 |
| 25. | Acción semántica símbolo if . . . . .                  | 36 |
| 26. | Acción semántica símbolo while . . . . .               | 36 |
| 27. | Acción semántica símbolo for . . . . .                 | 37 |
| 28. | Acción semántica símbolo case . . . . .                | 38 |
| 29. | Acción semántica símbolo valueList . . . . .           | 38 |
| 30. | Acción semántica símbolo value . . . . .               | 39 |
| 31. | Acción semántica símbolo cond . . . . .                | 39 |
| 32. | Acción semántica símbolo asgn . . . . .                | 40 |
| 33. | Acción semántica símbolo clear_screen . . . . .        | 41 |
| 34. | Acción semántica símbolo place . . . . .               | 41 |
| 35. | Acción semántica símbolo print . . . . .               | 42 |
| 36. | Acción semántica símbolo print_string . . . . .        | 42 |
| 37. | Acción semántica símbolo read . . . . .                | 43 |
| 38. | Acción semántica símbolo read_string . . . . .         | 43 |

|     |  |    |
|-----|--|----|
| 39. | Acción semántica símbolo repeat . . . . .            | 44 |
| 40. | Acción semántica símbolo exp I . . . . .             | 45 |
| 41. | Acción semántica símbolo exp II . . . . .            | 46 |
| 42. | Acción semántica símbolo exp III . . . . .           | 47 |
| 43. | Acción semántica símbolo exp IV . . . . .            | 48 |
| 44. | Acción semántica símbolo exp V . . . . .             | 49 |
| 45. | Acción semántica símbolo exp VI . . . . .            | 50 |
| 46. | Acción semántica símbolo listOfExp . . . . .         | 50 |
| 47. | Acción semántica símbolo restListOfExp . . . . .     | 51 |
| 48. | Representación de la clase Statement . . . . .       | 53 |
| 49. | Representación de la clase ExpNode . . . . .         | 58 |
| 50. | Representación de la jerarquía de ficheros . . . . . | 63 |
| 51. | Código del ejemplo binario.p . . . . .               | 65 |
| 52. | Código del ejemplo conversion.p . . . . .            | 66 |
| 53. | Código del ejemplo menu.p I . . . . .                | 67 |
| 54. | Código del ejemplo menu.p II . . . . .               | 68 |
| 55. | Código del ejemplo menu.p III . . . . .              | 69 |
| 56. | Código del ejemplo menu.p IV . . . . .               | 70 |
| 57. | Código del ejemplo menu.p V . . . . .                | 71 |
| 58. | Código del ejemplo menu.p VI . . . . .               | 72 |
| 59. | Código del ejemplo fibonnaci.p I . . . . .           | 74 |
| 60. | Código del ejemplo fibonnaci.p II . . . . .          | 75 |
| 61. | Código del ejemplo fibonnaci.p III . . . . .         | 76 |
| 62. | Código del ejemplo geometrico.p I . . . . .          | 78 |
| 63. | Código del ejemplo geometrico.p II . . . . .         | 79 |
| 64. | Código del ejemplo geometrico.p III . . . . .        | 80 |
| 65. | Código del ejemplo geometrico.p IV . . . . .         | 81 |

# **1. Introducción**

En este documento, vamos a realizar las explicaciones y análisis pertinentes al Trabajo Final de Prácticas de la asignatura, un interprete de pseudocódigo en español.

El trabajo consiste en usar los lenguajes de programación Flex para el análisis léxico y Bison para realizar el análisis sintáctico y semántico, apoyándonos en el código AST para este último.

Vamos a explicar el funcionamiento del propio lenguaje de pseudocódigo, tanto con respecto a componentes léxicos como a sentencias, el funcionamiento de la Tabla de Símbolos y del código AST.

Haremos un resumen de las funciones auxiliares codificadas, y una explicación sobre como la estructura de los directorios que componen el proyecto, así como una explicación de los modos de funcionamiento del interprete. Además, explicaremos como funciona el análisis Léxico y Sintáctico del intérprete. Por último, explicaremos los ejemplos entregados para probar el funcionamiento del intérprete y una conclusión y valoración del trabajo realizado.

Intentaremos centrarnos en la codificación lo menos posible, para hacer el documento lo mas legible posible.

## 2. Lenguaje de Pseudocódigo

El lenguaje de pseudocódigo es capaz de trabajar con dos elementos principales, componentes léxicos y sentencias.

### 2.1. Componentes léxicos

En cuanto a componentes léxicos, reconoce una gran variedad de tipos:

#### 2.1.1. Palabras reservadas

Se dividen en sentencias de control, constantes, operadores lógicos y componentes de manejo de pantallas.

| Tipo de componente léxico       | Componentes léxicos reconocidos   |
|---------------------------------|---|
| Sentencias de control           | <i>read, read_string,<br/>print, print_string,<br/>if, then, else, end_if<br/>while, do, end_while<br/>repeat, until,<br/>for, end_for,<br/>from, step, to<br/>case, value, default, end_case</i> |
| Constantes y operadores lógicos | <i>true, false,<br/>or, and, not</i>  |
| Manejo de pantalla              | <i>clear_screen, place</i>  |

Tabla 1: Componentes léxicos funcionales

### 2.1.2. Identificadores

Los identificadores del lenguaje se componen por una serie de letras, dígitos y subrayados, con la obligatoriedad de empezar únicamente por una letra, y no poder acabar en subrayados, o tener mas de un subrayado seguido.

Varios ejemplos de identificadores **válidos** serían:  
*variable*, *Identificador00*, *nombre\_persona1*, etc.

Varios ejemplos de identificadores **inválidos** serían:  
*variable\_*, *Identificador\_ \_ \_ 00*, *0\_nombre*, etc.

### 2.1.3. Números

Dentro del lenguaje se diferencian tres tipos de número, tratados todos de la misma forma:

**Números enteros:** Denotados por la expresión regular:

[0-9]+\\.?

**Números reales de punto fijo:** Denotados por la expresión regular:

[0-9]\*\\.[0-9]+

Ejemplos **válidos** serían: *0*, *0.25*, *14.28*, etc,

**Números reales con notación científica:** Denotados por la expresión regular:

[0-9]+(\\.[0-9]+)?[eE][+-]?[0-9]+

Ejemplos **válidos** serían: *2.345e-10*, *1.78E+5*, etc.

#### 2.1.4. Cadena

Las cadenas están delimitadas por comillas simples, y dentro de estas comillas simples, se reconoce cualquier conjunto de caracteres alfanuméricos, incluidos saltos de linea y tabulaciones.

Se regulan por la expresión regular:

```
"'([^\\"\\]|\"\\\",|\\n|\\t)*'"
```

Un ejemplo de cadena sería:

```
'Cadena de texto \n
de dos lineas'
```

#### 2.1.5. Operador de asignación

Para el operador de asignación, se ha usado el operador de Pascal:

`:=`

Es importante destacar que una variable puede cambiar de tipo en tiempo de ejecución, por ejemplo, pasando de un valor numérico a alfanuméricico.

Un ejemplo **válido** sería: *variable* `:=` 15

#### 2.1.6. Operadores aritméticos

Son funcionales distintos operadores aritméticos:

- Suma unaria y binaria: `+`
- Resta unaria y binaria: `-`
- Producto: `*`
- División: `/`
- División entera: `//`

- Módulo: %
- Potencia: ^

Cabe destacar que estas operaciones pueden ser aplicadas sobre variables con valores numéricos.

#### **2.1.7. Operador alfanumérico**

Solo hay un operador alfanumérico, el operador concatenación : ||

Un ejemplo de su uso sería:

```
nombre := 'José' || 'García';
```

#### **2.1.8. Operadores relacionales de números y cadenas**

Son funcionales distintos operadores relacionales, tanto para números como para cadenas:

- Menor que: <
- Menor o igual que: <=
- Mayor que: >
- Mayor o igual que: >=
- Igual que: =
- Distinto de: <>

Cabe destacar que estas operaciones pueden ser aplicadas sobre variables:

### **2.1.9. Operadores lógicos**

Solo se reconocen tres operadores lógicos:

- **Disyunción lógica:** or
- **Conjunción lógica:** and
- **Negación lógica:** not

Estas operaciones pueden combinarse o aplicarse sobre variables.

### **2.1.10. Comentarios**

En cuanto a comentarios, realizamos dos distinciones:

**Comentarios de una linea:** Lineas que comienzan por el carácter `#`. Un ejemplo sería:

```
#Comentario de una línea!
```

**Comentarios de varias lineas:** Lineas delimitadas por los símbolos de inicio « y final ». Un ejemplo sería:

```
<<Creado por:  
Valentín Avram & Victor Rojas >>
```

### **2.1.11. Punto y coma**

Para delimitar el final de una sentencia se utiliza el símbolo de control `;`. Un ejemplo de uso sería:

```
print(variable);
```

## 2.2. Sentencias

El lenguaje de pseudocódigo reconoce varias sentencias.

### 2.2.1. Asignación

En la asignación, se le da valor a una variable a través del operador asignación. Se le puede dar como valor a la variable una expresión numérica, alfanumérica u otras variables. La estructura de la asignación es:

```
identificador := expresion;
```

Algunos ejemplos de uso son:

```
minutos := segundos / 60;
nombre := 'Nombre' || 'Apellidos';
```

### 2.2.2. Lectura

En la lectura, diferenciamos la lectura de valores numéricos y valores alfanuméricos. La principal diferencia en cuanto a sintaxis son las palabras reservadas, *read* para valores numéricos y *read\_string* para valores alfanuméricos.

El funcionamiento es simplemente darle valor un valor introducido por pantalla por el usuario a la variable.

```
read(numero);
read_string(cadena);
```

Es importante que, aunque las cadenas empiecen y acaben por comillas simples, estas no se guardan como parte del valor de la variable alfanumérica.

### 2.2.3. Escritura

Al igual que en la lectura, en la escritura diferenciamos entre escritura de valores numéricos y alfanuméricos. Al igual que en el caso anterior, la principal diferencia en cuanto a sintaxis son las palabras reservadas, *print* para valores numéricos y *print\_string* para valores alfanuméricos.

Un ejemplo de uso sería:

```
print(numero);
print_string(nombreCompleto);
```

El funcionamiento es simplemente imprimir por pantalla la expresión o valor de la variable indicada. En el caso de valores alfanuméricos, es capaz de reconocer saltos de página y tabulaciones, sin tener en cuenta las comillas simples que delimitan las cadenas.

### 2.2.4. Comandos especiales

Se han desarrollado dos comandos especiales, para el control de la pantalla.

**Clear\_screen**, que a través de una macro se encarga de limpiar la pantalla de texto. Un ejemplo de su uso sería:

```
clear_screen;
```

**Place(entero, entero)**, que a través de una macro se encarga de ajustar la posición del lector de escritura en pantalla. Un ejemplo de su uso sería:

```
ejeX := 10;
ejeY := 20;

place(ejeX, ejeY);
```

## 2.3. Sentencias de control

Las sentencias de control son un tipo de sentencias mas complejas, pues llegan a controlar condiciones y actuar de una forma u otra según el cumplimiento de estas condiciones.

### **2.3.1. Sentencia condicional simple**

Comprueba que la condición se cumpla, y en ese caso, ejecuta uno o varios consecuentes. Un ejemplo de uso sería:

```
if (condicion = true) then print_string('Condición cierta!'); end_if;
```

### **2.3.2. Sentencia condicional compuesta**

Similar a la sentencia condicional simple, pero permite ejecutar consecuentes en caso que la condición no se cumpla. Un ejemplo sería:

```
if(condicion = true) then print_string('Condición cierta!');  
else print_string('Condición no cumplida!'); end_if;
```

### **2.3.3. Bucle while**

Mientras la condición se cumpla, ejecutará la misma serie de instrucciones, hasta que la condición deje de cumplirse. Un ejemplo sería:

```
while (contador < 10) do  
    print_string('Entramos en el bucle!');  
    print(contador);  
    contador = contador + 1;  
end_while;
```

### **2.3.4. Bucle repeat**

El bucle repeat repetirá la misma serie de instrucciones hasta cumplir la condición. Un ejemplo sería:

```
repeat  
    print_string('Iteración del bucle!');  
    control = control + 1;  
until (control > 10);
```

### 2.3.5. Bucle for

El bucle for ejecutará una serie de instrucciones en un bucle, que irá desde una expresión numérica hasta otra, teniendo la opción de avanzar entre ese intervalo de expresiones en base a un cierto paso. Un ejemplo sería:

```
for contador from 0 to 10 step 2 do
    print_string('Iteramos de dos en dos!');
end_for;
```

Se debe recalcar que el *step*, el paso que damos al iterar, es opcional, y en caso de no indicarse se iterará de uno en uno. Además, en caso de que la variable sobre la que se itera no exista, se crea con un valor predeterminado, el valor numérico cero.

### 2.3.6. Estructura case

La sentencia case comprueba el estado de una variable sobre distintos posibles valores de una variable o expresión, y dependiendo de ese valor, se ejecutará una instrucción u otra. Un ejemplo de uso sería:

```
case(expresionNumerica)
    value 1: print_string('Primer posible valor');
    value 10: print_string('Segundo posible valor');
end_case;
```

### 3. Tabla de símbolos

La tabla de Símbolos se puede visualizar en la tabla 2.

Tabla 2: Tabla de Símbolos

| TOKEN              | ID  |
|--------------------|-----|
| SEMICOLON          | 258 |
| PRINT              | 259 |
| READ               | 260 |
| IF                 | 261 |
| ELSE               | 262 |
| WHILE              | 263 |
| FOR                | 264 |
| PRINT_STRING       | 265 |
| READ_STRING        | 266 |
| THEN               | 267 |
| END_IF             | 268 |
| DO                 | 269 |
| END WHILE          | 270 |
| REPEAT             | 271 |
| UNTIL              | 272 |
| END FOR            | 273 |
| FROM               | 274 |
| STEP               | 275 |
| TO                 | 276 |
| CASE               | 277 |
| VALUE              | 278 |
| AND                | 279 |
| NOT                | 280 |
| OR                 | 281 |
| CLEAR_SCREEN_TOKEN | 282 |
| PLACE_TOKEN        | 283 |
| TYPE_OF            | 284 |
| END_CASE           | 285 |
| DEFAULT            | 286 |
| LEFTCURLYBRACKET   | 287 |
| RIGHTCURLYBRACKET  | 288 |
| ASSIGNMENT         | 289 |
| COMMA              | 290 |

Continúa en la siguiente página

Tabla 2 – Continuación de la página anterior

| TOKEN            | ID  |
|------------------|-----|
| TWO_DOTS         | 291 |
| NUMBER           | 292 |
| BOOL             | 293 |
| STRING           | 294 |
| VARIABLE         | 295 |
| UNDEFINED        | 296 |
| CONSTANT         | 297 |
| BUILTIN          | 298 |
| KEYWORD          | 299 |
| GREATER_OR_EQUAL | 300 |
| LESS_OR_EQUAL    | 301 |
| GREATER_THAN     | 302 |
| LESS_THAN        | 303 |
| EQUAL            | 304 |
| NOT_EQUAL        | 305 |
| PLUS             | 306 |
| MINUS            | 307 |
| CONCATENATION    | 308 |
| MULTIPLICATION   | 309 |
| DIVISION         | 310 |
| MODULO           | 311 |
| INT_DIVISION     | 312 |
| LPAREN           | 313 |
| RPAREN           | 314 |
| UNARY            | 315 |
| POWER            | 316 |

Para la implementación de la tabla de símbolos se han usado las siguientes clases, visualizadas en la figura 1.

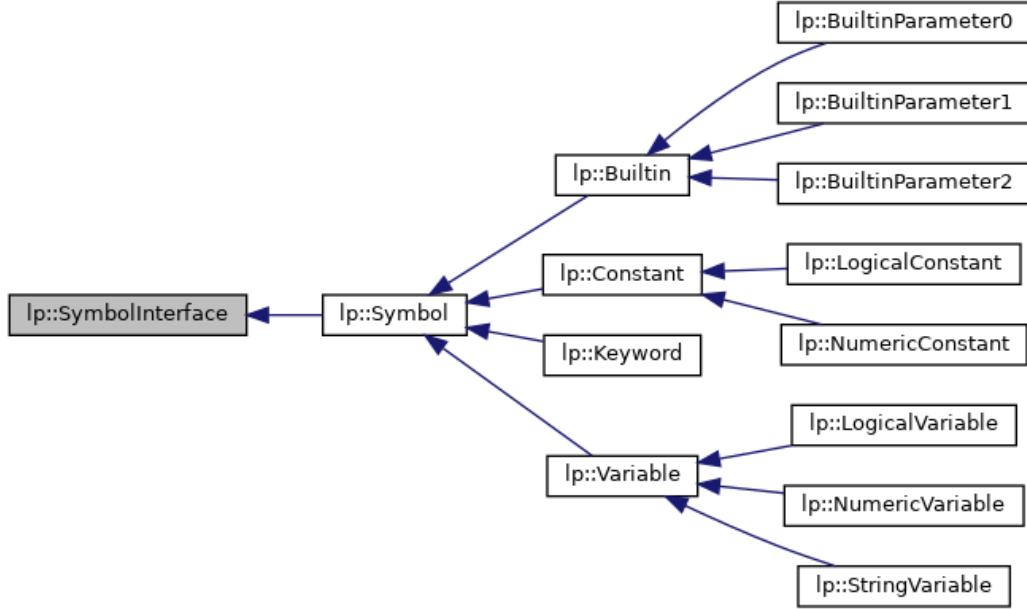


Figura 1: Jerarquía de las clases de la tabla símbolos

En el esquema 1 se muestra como todas las clases heredan de la clase Symbol, la cual hereda de SymbolInterface. También se observa como los símbolos son implementados mediante Builtin, Constant, Keyword y Variable. Dichas funciones se encontraban en el ejemplo 17 de las prácticas de Bison

Builtin cuenta con 3 clases:

1. **BuiltinParameter0**: clase que implementa las funciones de 0 parámetros.
2. **BuiltinParameter1**: clase que implementa las funciones con 1 parámetro.
3. **BuiltinParameter2**: clase que implementa las funciones con 2 parámetros.

Constant cuenta con 2 clases:

1. **LogicalConstant**: clase que implementa las constantes lógicas.
2. **NumericConstant**: clase que implementa las constantes numéricas.

Variable cuenta con 3 clases:

1. **LogicalVariable**: clase que implementa las variables de tipo lógico.
2. **NumericVariable**: clase que implementa las variables de tipo numérico.
3. **StringVariable**: clase que implementa las variables de tipo cadena

La clase StringVariable la hemos implementado en nuestro intérprete, siendo una clase encargada de almacenar valores alfanuméricos, usando el tipo String de C++. Esta implementación puede visualizarse en la imagen 2.

```
● ○ ●
class StringVariable : public lp::Variable
{
private:
    std::string _value; //!< Name of the variable

public:

    inline void setValue(std::string value)
    {
        this->_value = value;
    }

    inline StringVariable(std::string name="", int token = 0, int
type = 0, std::string value = "") : Variable(name,token,type)
    {
        this->setValue(value);
    }

    StringVariable(const StringVariable &s)
    {
        this->setName(s.getName());
        this->setToken(s.getToken());
        this->setValue(s.getValue());
    }

    inline std::string getValue() const
    {
        return this->_value;
    }

    void read();

    void write() const;

    StringVariable &operator=(const StringVariable &s);

    friend std::istream &operator>>(std::istream &is, StringVariable
&s);

    friend std::ostream &operator<<(std::ostream &os, const
StringVariable &s);
};


```

Figura 2: Declaración de la clase StringVariable

## 4. Análisis Léxico

En cuanto al análisis léxico, podemos mostrar todas las declaraciones y reglas normales para el reconocimiento de componentes léxicos.

### 4.1. Zona de declaraciones

Las declaraciones se pueden visualizar en la figura 3.



The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays a series of regular expression definitions for lexical components. The definitions include:

- `/*! \name REGULAR DEFINITIONS */`
- `DIGIT [0-9]`
- `LETTER [a-zA-Z]`
- `/*! TODO: NUMBER IN SCINOT AND ITS ERROR*/`
- `NUMBER1 {DIGIT}+\.?`
- `NUMBER2 {DIGIT}*\.{DIGIT}+`
- `SCIENTIFIC_NOTATION {DIGIT}+(\.{DIGIT}+)?((e|E)[+-]?)?{DIGIT}+)?`
- `/*! NEW in v0.13*/`
- `NUMBER_WRONG1 {DIGIT}+\.\.+{DIGIT}*`
- `NUMBER_WRONG2 {DIGIT}+({e|E})({e|E})+{DIGIT}*`
- `NUMBER_WRONG3 {DIGIT}+({e|E})({\+|\-\-})({\+|\-\-})+{DIGIT}*`
- `IDENTIFIER {LETTER}({LETTER}|{DIGIT})|_({LETTER}|{DIGIT}))*`
- `/*! NEW in v0.13*/`
- `IDENTIFIER_WRONG1 _({LETTER}|{DIGIT})|_({LETTER}|{DIGIT}))*`
- `IDENTIFIER_WRONG2 {LETTER}({LETTER}|{DIGIT})|_({LETTER}|{DIGIT}))*_`
- `IDENTIFIER_WRONG3 {LETTER}({LETTER}|{DIGIT})|__({LETTER}|{DIGIT}))+`
- `CHARSTRING """([^\\\\"]|\"\\\"|\\n|\\t)*"""`

Figura 3: Declaraciones básicas del análisis léxico

Además, usamos dos estados que, por separado, evalúan errores léxicos o comentarios compuestos de varias líneas. Se pueden visualizar en la figura 4.



Figura 4: Estados usados en el análisis léxico

## 4.2. Reglas normales

A continuación, mostraremos todas las reglas usadas para realizar el análisis léxico. Mostraremos la codificación de las reglas y explicaremos brevemente su funcionamiento.

En la figura 5 podemos ver las dos primeras expresiones regulares, que ignoran la lectura de saltos de línea o tabulaciones.

A screenshot of a terminal window with a dark background. At the top, there are three colored circles: red, yellow, and green. Below them, the text is displayed in white. It shows two regular expression rules. The first rule is [ \t] { ; } /\* skip white space and tabular \*/. The second rule is \n { /\* Line counter \*/ lineNumber++; /\* MODIFIED in example 3 \*/ /\* COMMENTED in example 5 \*/ /\* return NEWLINE; \*/ }

Figura 5: Análisis léxico. Reglas I

En las figuras 6, 7, 8, 9 podemos ver las expresiones regulares que de-  
notan las palabras reservadas en nuestro lenguaje, sin hacer distinción entre  
mayúsculas y minúsculas.

```
● ● ●  
(?i:read)  {      return READ;  
}  
  
(?i:read_string)  {      return READ_STRING;  
}  
  
(?i:print)  {      return PRINT;  
}
```

Figura 6: Análisis léxico. Reglas II

```
● ● ●  
(?i:print_string)  {      return PRINT_STRING;  
}  
  
(?i:clear_screen)  {      return CLEAR_SCREEN_TOKEN;  
}  
  
(?i:place)  {      return PLACE_TOKEN;  
}  
  
(?i:default)  {      return DEFAULT;  
}  
  
(?i:end_case)  {      return END_CASE;  
}  
  
(?i:type_of)  {      return TYPE_OF;  
}
```

Figura 7: Análisis léxico. Reglas III

```
● ● ●  
(?i:for) { return FOR;  
}  
  
(?i:if) { return IF;  
}  
  
(?i:then) { return THEN;  
}  
  
(?i:end_if) { return END_IF;  
}  
  
(?i:else) { return ELSE;  
}  
  
(?i:while) { return WHILE;  
}  
  
(?i:do) { return DO;  
}  
  
(?i:end_while) { return END WHILE;  
}  
  
(?i:repeat) { return REPEAT;  
}  
  
(?i:until) { return UNTIL;  
}  
  
(?i:end_for) { return END FOR;  
}
```

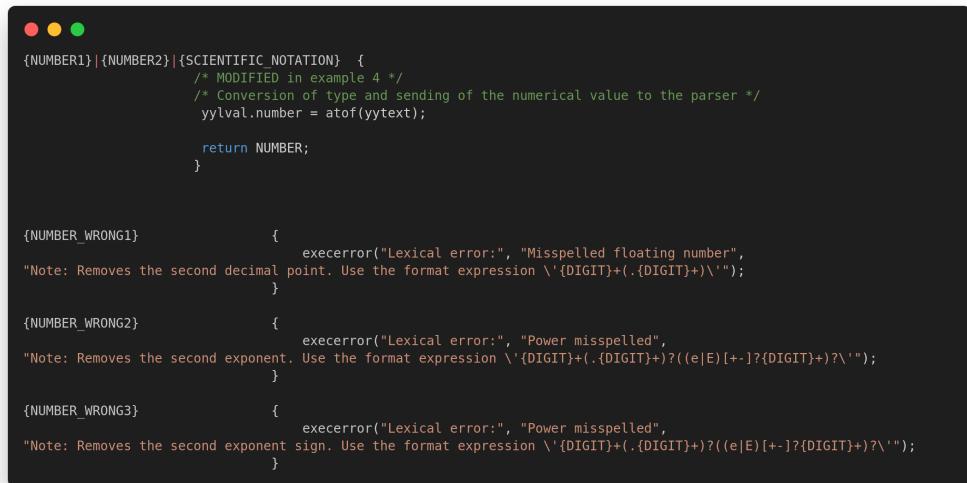
Figura 8: Análisis léxico. Reglas IV



```
(?i:from) { return FROM; }
(?i:step) { return STEP; }
(?i:to) {
    return TO;
}
(?i:case) { return CASE; }
(?i:value) { return VALUE; }
(?i:and) { return AND; }
(?i:not) { return NOT; }
(?i:or) { return OR; }
```

Figura 9: Análisis léxico. Reglas V

En la figura 10 vemos las reglas para el reconocimiento de números enteros, reales, en notación científica, así como algunos posibles fallos léxicos.



```
/* MODIFIED in example 4 */
/* Conversion of type and sending of the numerical value to the parser */
yyval.number = atof(yytext);

return NUMBER;
}

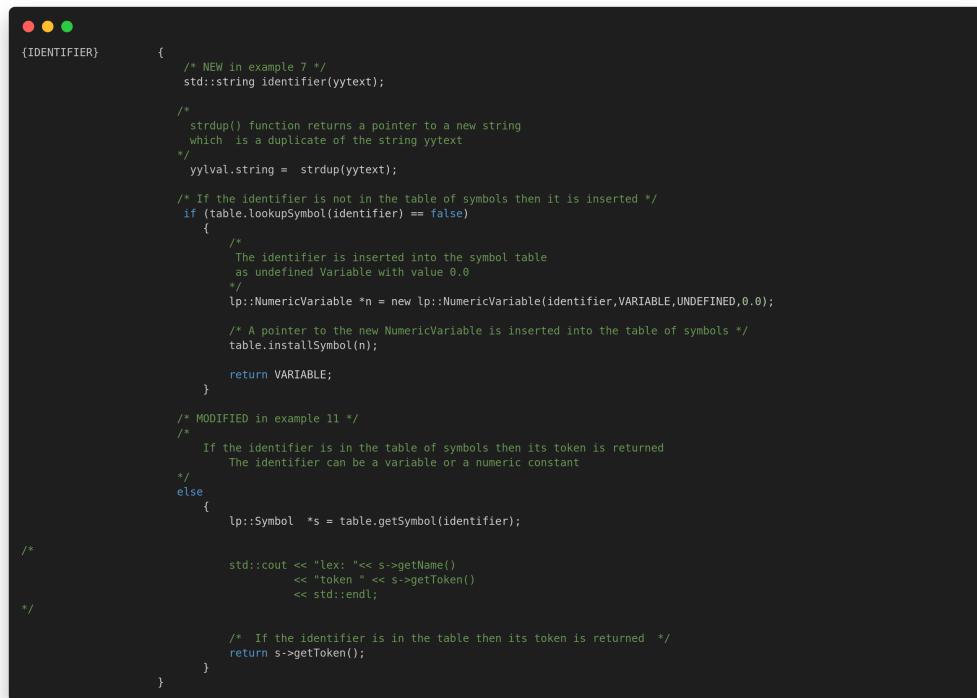
{NUMBER_WRONG1}
{
    execerror("Lexical error:", "Misspelled floating number",
"Note: Removes the second decimal point. Use the format expression '\{DIGIT\}+\{DIGIT\}\''");
}

{NUMBER_WRONG2}
{
    execerror("Lexical error:", "Power misspelled",
"Note: Removes the second exponent. Use the format expression '\{DIGIT\}+\{DIGIT\}+?((e|E)[+-]\{DIGIT\}+)?\''");
}

{NUMBER_WRONG3}
{
    execerror("Lexical error:", "Power misspelled",
"Note: Removes the second exponent sign. Use the format expression '\{DIGIT\}+\{DIGIT\}+?((e|E)[+-]\{DIGIT\}+)?\''");
}
```

Figura 10: Análisis léxico. Reglas VI

En la figura 11 vemos la reglas para el reconocimiento de identificadores, y su tratamiento respecto a la tabla de símbolos. Además, en la figura 12 se pueden visualizar el reconocimiento de identificadores incorrectos.



```

/* IDENTIFIER */
{
    /* NEW in example 7 */
    std::string identifier(yytext);

    /*
     * strdup() function returns a pointer to a new string
     * which is a duplicate of the string yytext
     */
    yylval.string = strdup(yytext);

    /* If the identifier is not in the table of symbols then it is inserted */
    if (table.lookupSymbol(identifier) == false)
    {
        /*
         * The identifier is inserted into the symbol table
         * as undefined Variable with value 0.0
         */
        lp::NumericVariable *n = new lp::NumericVariable(identifier,VARIABLE,UNDEFINED,0.0);

        /* A pointer to the new NumericVariable is inserted into the table of symbols */
        table.installSymbol(n);

        return VARIABLE;
    }

    /* MODIFIED in example 11 */
    /*
     * If the identifier is in the table of symbols then its token is returned
     * The identifier can be a variable or a numeric constant
     */
    else
    {
        lp::Symbol *s = table.getSymbol(identifier);

        /*
         * std::cout << "lex: " << s->getName()
         * << "token " << s->getToken()
         * << std::endl;
        */

        /* If the identifier is in the table then its token is returned */
        return s->getToken();
    }
}

```

Figura 11: Análisis léxico. Reglas VII



```

/* IDENTIFIER_WRONG1 */
{
    execerror("Lexical error: Incorrectly spelled identifier", "The identifier cannot begin with an underscore",
    "Note: delete the first underscore");
}

/* IDENTIFIER_WRONG2 */
{
    execerror("Lexical error: Incorrectly spelled identifier", "The identifier cannot end with an underscore",
    "Note: delete the last underscore");
}

/* IDENTIFIER_WRONG3 */
{
    execerror("Lexical error: Incorrectly spelled identifier", "The identifier cannot have double underscores",
    "Note: delete one underscore");
}

/* CHARSTRING */
{
    memmove(yytext, yytext+1, strlen(yytext));
    yytext[strlen(yytext)-1] = '\0';
    yylval.string = strdup(yytext);

    return STRING;
}

```

Figura 12: Análisis léxico. Reglas VIII

En la figura 13 vemos las reglas para el reconocimiento de operadores numéricos.

```
● ● ●
"-"
{ return MINUS; }          /* NEW in example 3 */

"+"
{ return PLUS; }           /* NEW in example 3 */

"**"
{ return MULTIPLICATION; } /* NEW in example 3 */

"/"
{ return DIVISION; }        /* NEW in example 3 */

"//"
{ return INT_DIVISION; }

"("
{ return LPAREN; }          /* NEW in example 3 */

")"
{ return RPAREN; }           /* NEW in example 3 */
```

Figura 13: Análisis léxico. Reglas IX

En la figura 14 vemos las reglas para el reconocimiento de operadores relacionales.

```
● ● ●
"="
{ return EQUAL; }           /* NEW in example 15 */

"<>"
{ return NOT_EQUAL; }        /* NEW in example 15 */

">="
{ return GREATER_OR_EQUAL; } /* NEW in example 15 */

"><="
{ return LESS_OR_EQUAL; }    /* NEW in example 15 */

">>"
{ return GREATER_THAN; }     /* NEW in example 15 */

"><"
{ return LESS_THAN; }        /* NEW in example 15 */

"!"
{ return NOT; }              /* NEW in example 15 */

"||"
{ return CONCATENATION; }

"&&"
{ return AND; }               /* NEW in example 15 */

"{"
{ return LETFCURLYBRACKET; } /* NEW in example 17 */

"}"
{ return RIGHTCURLYBRACKET; } /* NEW in example 17 */

":"
{ return TWO_DOTS; }          /* V 0.9 */
```

Figura 14: Análisis léxico. Reglas X

En la figura 15 vemos las reglas para el reconocimiento de comentarios de una sola línea, y el tratamiento léxico de estados de varios comentarios, usando estados.



```
#.*    { printf("Commentario: %s\n", yytext); } /* V 0.9*/
"<<"    {
    BEGIN(COMMENTS);
    yymore();
}

<COMMENTS>"<"  {
    execerror("Nested comments are not allowed", yytext);
    BEGIN(INITIAL);
}

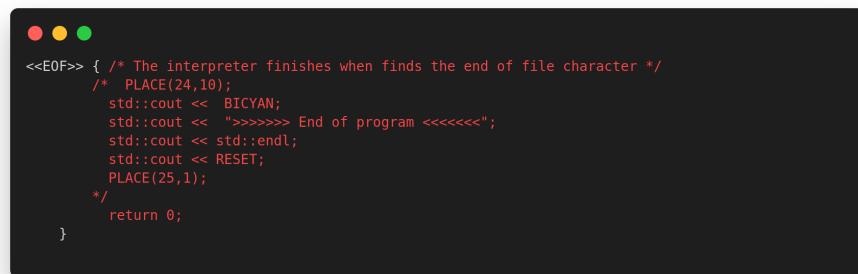
<COMMENTS>[^>>\n]  {
    yymore();
}

<COMMENTS>\n  {
    lineNumber++;
    yymore();
}
<COMMENTS>">>"  {
    BEGIN(INITIAL);
}

<COMMENTS><<EOF>  {
    execerror("End of file found in comment", "You must close the comment"
);
    return 0;
}
```

Figura 15: Análisis léxico. Reglas XI

En las figuras 16 y 17 se pueden visualizar el tratamiento de errores usando estados y de final de la archivo, aunque realmente no se haga nada al finalizar el archivo.

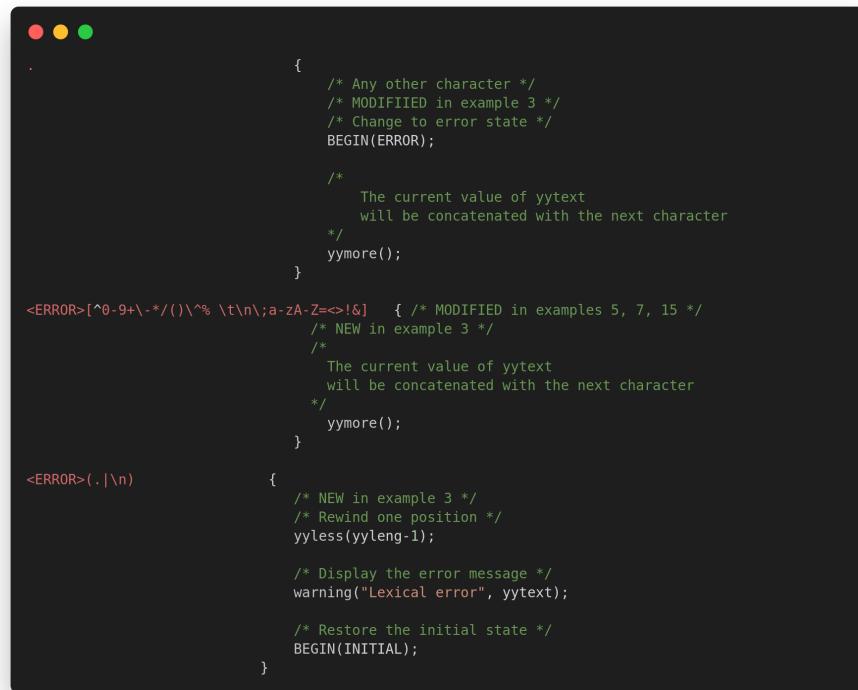


```


<<EOF>> { /* The interpreter finishes when finds the end of file character */
    /* PLACE(24,10);
    std::cout << BICYAN;
    std::cout << ">>>>> End of program <<<<<<";
    std::cout << std::endl;
    std::cout << RESET;
    PLACE(25,1);
    */
    return 0;
}


```

Figura 16: Análisis léxico. Reglas XII



```


{
    /* Any other character */
    /* MODIFIED in example 3 */
    /* Change to error state */
    BEGIN(ERROR);

    /*
        The current value of yytext
        will be concatenated with the next character
    */
    yymore();
}

<ERROR>[^0-9+\-*()^% \t\n\;a-zA-Z=<>!&] { /* MODIFIED in examples 5, 7, 15 */
    /* NEW in example 3 */
    /*
        The current value of yytext
        will be concatenated with the next character
    */
    yymore();
}

<ERROR>(.|\n) {
    /* NEW in example 3 */
    /* Rewind one position */
    yyless(yylen-1);

    /* Display the error message */
    warning("Lexical error", yytext);

    /* Restore the initial state */
    BEGIN(INITIAL);
}


```

Figura 17: Análisis léxico. Reglas XIII

## 5. Análisis Sintáctico

En cuanto al análisis sintáctico, podemos nombrar y explicar los símbolos terminales y no terminales, comentar las reglas de producción de la gramática y analizar las acciones semánticas realizadas.

### 5.1. Símbolos terminales

Los símbolos terminales explicados son los siguientes:

1. **SEMICOLON**: token ';' cuya función es separar distintos Statements
2. **PRINT**: token para la impresión de datos numéricos
3. **PRINT\_STRING**: token para la impresión de datos alfanuméricos
4. **READ**: token 'read' para la lectura de datos numéricos
5. **READ\_STRING**: token para la lectura de datos alfanuméricos
6. **IF, ELSE, THEN, END\_IF**: tokens para las sentencias condicionales
7. **WHILE, DO, END WHILE**: tokens para la función iterativa 'while'
8. **FOR, FROM, TO, STEP, DO, END FOR**: tokens para la función iterativa 'for'
9. **REPEAT, UNTIL**: token para la función iterativa 'repeat'
10. **TYPE\_OF**: token para la impresión de datos de tipo dinámico
11. **CASE, VALUE, DEFAULT, END\_CASE**: tokens para la estructura condicional 'case'
12. **CLEAR\_SCREEN\_TOKEN, PLACE\_TOKEN**: tokens para la función 'clear\_screen' la cual limpia la terminal y 'place' la cual permite posicionar el cursor
13. **LEFTCURLYBRACKET, RIGTHCURLYBRACKET**: tokens '{' y '}'

14. **ASSIGMENT**: token para la asignación ':='
15. **COMMA, TWO\_DOTS**: tokens para la coma ',' y para los dos puntos ':'
16. **NUMBER, STRING, BOOL, UNDEFINED**: tokens para los tipos que contendrá nuestro lenguaje, siendo estos: número, cadena, booleano, e indefinido
17. **VARIABLE**: token para variables
18. **OR, AND, NOT**: token para el 'o' lógico, 'y' lógico y para la negación lógica
19. **GREATER\_OR\_EQUAL, LESS\_OR\_EQUAL, GREATER\_THAN, LESS\_THAN, EQUAL, NOT\_EQUAL**: tokens usados para la comparación
20. **PLUS, MINUS, CONCATENATION**: tokens para la suma y resta aritmética y el operador concatenación para alfanuméricos
21. **MULTIPLICATION, DIVISION, MODULO, INT\_DIVISION**: tokens para la multiplicación, la división, el módulo y la división entera
22. **POWER**: token para la operación potencia
23. **LPRAREN, RPAREN**: tokens para el paréntesis izquierdo '(' y para el paréntesis derecho ')'.

## 5.2. Símbolos no terminales

Los símbolos no terminales explicados son los siguientes:

1. **program**: símbolo inicial de nuestra gramática
2. **exp**: token que representa una expresión
3. **ListofExp, restListOfExp**: tokens que representan listas de expresiones
4. **type\_of**: token que representa la estructura usada por una función 'type\_of'
5. **stmt, block**: token que representa una acción
6. **stmtlist**: token que representa una lista de acciones
7. **if, cond**: tokens que representan la expresión condicional 'if' y la condición
8. **while**: token que representa la función iterativa 'while'
9. **for**: token que representa la función iterativa 'for'
10. **repeat**: token que representa la función iterativa 'repeat'
11. **case**: token que representa la estructura 'case' completa
12. **value, defaultValue**: tokens que representan los casos individuales de la estructura 'case' y el caso por defecto
13. **valuelist**: token que representa una lista de casos para la estructura 'case'
14. **asgn**: token que representa una asignación
15. **clear\_screen**: token que representa la estructura usada para limpiar la terminal
16. **place**: token que representa la estructura usada para posicionar el puntero
17. **print, print\_string**: tokens que representan las funciones usadas para imprimir números y cadenas por terminal
18. **read, read\_string**: tokens que representan las funciones usadas para la asignación interactiva de números y cadenas

### 5.3. Reglas de producción de la gramática

Las reglas de producción que generan el lenguaje son las siguientes:

- 1) **program** → **stmtlist**
- 2) **stmtlist** →  $\epsilon$
- 3) **stmtlist** → **stmt**
- 4) **stmtlist** → **error**
- 5) **stmt** → *SEMICOLON*
- 6) **stmt** → **asgn SEMICOLON**
- 7) **stmt** → **print SEMICOLON**
- 8) **stmt** → **print\_string SEMICOLON**
- 9) **stmt** → **read SEMICOLON**
- 10) **stmt** → **read\_string SEMICOLON**
- 11) **stmt** → **if SEMICOLON**
- 12) **stmt** → **while SEMICOLON**
- 13) **stmt** → **block SEMICOLON**
- 14) **stmt** → **for SEMICOLON**
- 15) **stmt** → **clear\_screen SEMICOLON**
- 16) **stmt** → **repeat SEMICOLON**
- 17) **stmt** → **place SEMICOLON**
- 18) **stmt** → **case SEMICOLON**
- 19) **stmt** → **type\_of SEMICOLON**
- 20) **type\_of** → *TYPE\_OF LPAREN exp RPAREN*
- 21) **block** → *LEFTCURLYBRACKET stmtlist RIGHTCURLYBRACKET*
- 22) **controlSymbol** →  $\epsilon$
- 23) **if** → *IF controlSymbol cond THEN stmtlist END\_IF*
- 24) **if** → *IF controlSymbol cond THEN stmtlist ELSE stmtlist END\_IF*
- 25) **while** → *WHILE controlSymbol cond DO stmtlist END WHILE*
- 26) **for** → **FOR controlSymbol VARIABLE**  
          **FROM exp TO exp STEP exp**  
          **DO stmtlist END\_FOR**
- 27) **for** → **FOR controlSymbol VARIABLE**  
          **FROM exp TO exp DO stmtlist END\_FOR**
- 28) **case** → **CASE controlSymbol LPAREN exp RPAREN**  
          **valuelist END\_CASE**

- 29) **case** → CASE controlSymbol LPAREN exp RPAREN  
     valuelist defaultValue END\_CASE
- 30) **valuelist** →  $\epsilon$   
 31) **valuelist** → **valuelist value**  
 32) **valuelist** → *VALUE NUMBER TWO\_DOTS stmtlist*  
 33) **valuelist** → *VALUE CONSTANT TWO\_DOTS stmtlist*  
 34) **valuelist** → *VALUE STRING TWO\_DOTS stmtlist*  
 35) **defaultValue** → *DEFAULT TWO\_DOTS stmtlist*  
 36) **cond** → LPAREN exp RPAREN  
 37) **asgn** → VARIABLE ASSIGNMENT exp  
 38) **asgn** → VARIABLE ASSIGNMENT asgn  
 39) **asgn** → CONSTANT ASSIGNMENT exp  
 40) **asgn** → CONSTANT ASSIGNMENT asgn  
 41) **asgn** → KEYWORD ASSIGNMENT exp  
 42) **clear\_screen** → CLEAR SCREEN TOKEN SEMICOLON  
 43) **place** → PLACE TOKEN LPAREN exp COMMA exp RPAREN SEMICOLON  
 44) **print** → PRINT exp  
 45) **print\_string** → PRINT\_STRING exp  
 46) **read** → READ LPAREN VARIABLE RPAREN  
 47) **read** → READ LPAREN CONSTANT RPAREN  
 48) **read\_string** → READ\_STRING LPAREN VARIABLE RPAREN  
 49) **repeat** → REPEAT controlSymbol stmtlist UNTIL cond SEMICOLON  
 50) **exp** → NUMBER  
 51) **exp** → STRING  
 52) **exp** → **exp PLUS exp**  
 53) **exp** → **exp MINUS exp**  
 54) **exp** → **exp MULTIPLICATION exp**  
 55) **exp** → **exp DIVISION exp**  
 56) **exp** → **exp INT\_DIVISION exp**  
 57) **exp** → LPAREN exp RPAREN  
 58) **exp** → PLUS exp  
 59) **exp** → MINUS exp  
 60) **exp** → **exp MODULO exp**  
 61) **exp** → **exp POWER exp**  
 62) **exp** → VARIABLE  
 63) **exp** → CONSTANT  
 64) **exp** → BUILTIN LPAREN listOfExp RPAREN  
 65) **exp** → **exp GREATER\_THAN exp**  
 66) **exp** → **exp GREATER\_OR\_EQUAL exp**

```

67)exp → exp LESS_THAN exp
68)exp → exp LESS_OR_EQUAL exp
69)exp → exp EQUAL exp
70)exp → exp NOT_EQUAL exp
71)exp → exp AND exp
72)exp → exp CONCATENATION exp
73)exp → exp OR exp
74)exp → NOT exp
75)listOfExp → ε
76)listOfExp → exp restListOfExp
77)restListOfExp → ε
78)restListOfExp → COMMA exp restListOfExp

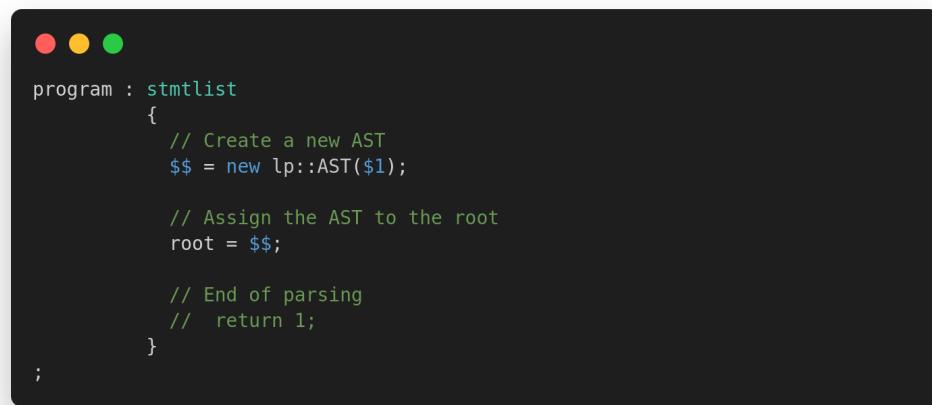
```

## 5.4. Acciones semánticas

A continuación, mostraremos el código correspondiente a las acciones semánticas, que no son más que las representaciones de las reglas de producción de la gramática.

### 5.4.1. program

La acción semántica del símbolo no terminal **program** se puede visualizar en la figura 18.



```

program : stmtlist
{
    // Create a new AST
    $$ = new lp::AST($1);

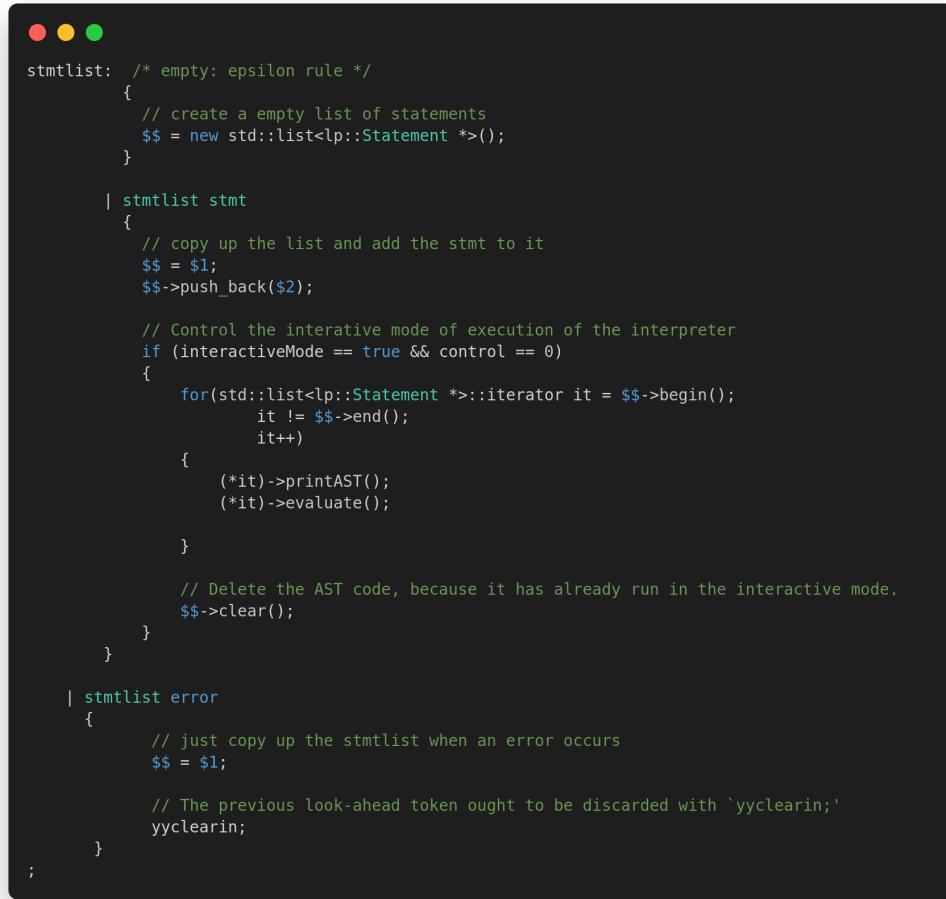
    // Assign the AST to the root
    root = $$;

    // End of parsing
    // return 1;
}
;
```

Figura 18: Acción semántica símbolo program

#### 5.4.2. stmtlist

La acción semántica del símbolo no terminal **stmtlist** se puede visualizar en la figura 19.



```
stmtlist: /* empty: epsilon rule */
{
    // create a empty list of statements
    $$ = new std::list<lp::Statement *>();
}

| stmtlist stmt
{
    // copy up the list and add the stmt to it
    $$ = $1;
    $$->push_back($2);

    // Control the interative mode of execution of the interpreter
    if (interactiveMode == true && control == 0)
    {
        for(std::list<lp::Statement *>::iterator it = $$->begin();
            it != $$->end();
            it++)
        {
            (*it)->printAST();
            (*it)->evaluate();
        }

        // Delete the AST code, because it has already run in the interactive mode.
        $$->clear();
    }
}

| stmtlist error
{
    // just copy up the stmtlist when an error occurs
    $$ = $1;

    // The previous look-ahead token ought to be discarded with `yyclearin;`
    yyclearin;
}
;
```

Figura 19: Acción semántica símbolo stmtlist

#### 5.4.3. stmt

La acción semántica del símbolo no terminal **stmt** se puede visualizar en las figuras 20 y 21.

```
stmt: SEMICOLON /* Empty statement: ";" */
{
    // Create a new empty statement node
    $$ = new lp::EmptyStmt();
}
| asgn SEMICOLON
{
    // Default action
    // $$ = $1;
}
| print SEMICOLON
{
    // Default action
    // $$ = $1;
}
| print_string SEMICOLON
{
    // Default action
    // $$ = $1;
}
| read SEMICOLON
{
    // Default action
    // $$ = $1;
}
| read_string SEMICOLON
{
    // Default action
    // $$ = $1;
}
/* NEW in example 17 */
| if
{
    // Default action
    // $$ = $1;
}
/* NEW in example 17 */
| while
{
    // Default action
    // $$ = $1;
}
/* NEW in example 17 */
| block
{
    // Default action
    // $$ = $1;
}
```

Figura 20: Acción semántica símbolo stmt I



```

/* NEW in version 0.4 */
| for{
    // Default action
    // $$ = $1;
}
/* NEW in version 0.5 */
| clear_screen
{
    // Default action
    // $$ = $1;
}
/* NEW in version 0.6 */
| repeat
{
    //Default action
    // $$ = $1;
}
| place
{
    //Default action
    // $$ = $1;
}

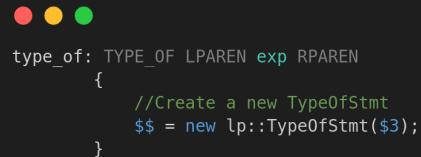
/*NEW in version 0.11*/
| case
{
    //Default action
    // $$ = $1;
}

/*NEW in version 0.11*/
| type_of
{
    //Default action
    // $$ = $1;
}
;
```

Figura 21: Acción semántica símbolo stmt II

#### 5.4.4. type\_of

La acción semántica del símbolo terminal **type\_of** se puede visualizar en la figura 22.



```

type_of: TYPE_OF LPAREN exp RPAREN
{
    //Create a new TypeOfStmt
    $$ = new lp::TypeOfStmt($3);
}
```

Figura 22: Acción semántica símbolo type\_of

#### 5.4.5. block

La acción semántica del símbolo no terminal **block** se puede visualizar en la figura 23.

```
● ● ●
block: LEFTCURLYBRACKET stmtlist RIGHTCURLYBRACKET
{
    // Create a new block of statements node
    $$ = new lp::BlockStmt($2);
}
;
```

Figura 23: Acción semántica símbolo block

#### 5.4.6. controlSymbol

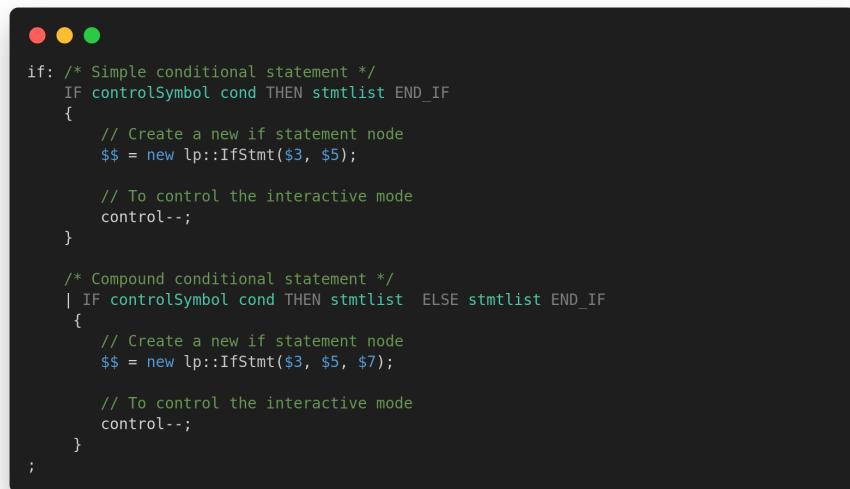
La acción semántica del símbolo no terminal **controlSymbol** se puede visualizar en la figura 24.

```
● ● ●
controlSymbol: /* Epsilon rule*/
{
    // To control the interactive mode in "if" and "while" sentences
    control++;
}
;
```

Figura 24: Acción semántica símbolo controlSymbol

#### 5.4.7. if

La acción semántica del símbolo terminal **if** se puede visualizar en la figura 25.



```
if: /* Simple conditional statement */
| IF controlSymbol cond THEN stmtlist END_IF
{
    // Create a new if statement node
    $$ = new lp::IfStmt($3, $5);

    // To control the interactive mode
    control--;
}

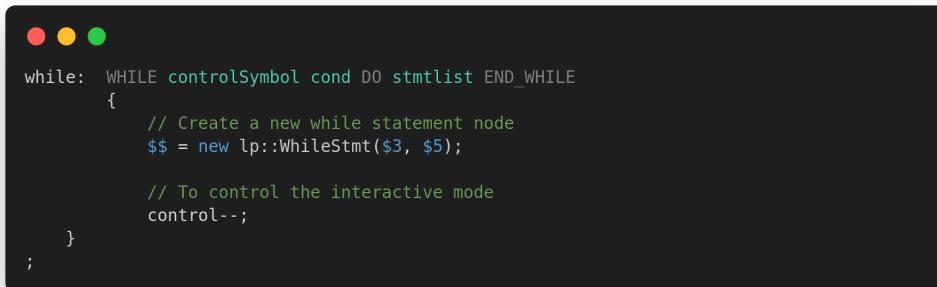
/* Compound conditional statement */
| IF controlSymbol cond THEN stmtlist ELSE stmtlist END_IF
{
    // Create a new if statement node
    $$ = new lp::IfStmt($3, $5, $7);

    // To control the interactive mode
    control--;
}
;
```

Figura 25: Acción semántica símbolo if

#### 5.4.8. while

La acción semántica del símbolo terminal **while** se puede visualizar en la figura 26.



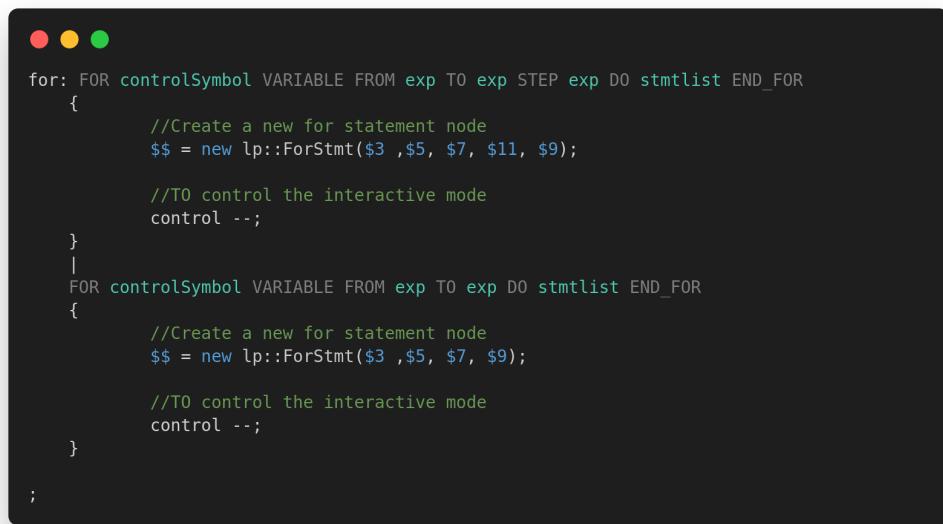
```
while: WHILE controlSymbol cond DO stmtlist END WHILE
{
    // Create a new while statement node
    $$ = new lp::WhileStmt($3, $5);

    // To control the interactive mode
    control--;
}
;
```

Figura 26: Acción semántica símbolo while

### 5.4.9. for

La acción semántica del símbolo terminal **for** se puede visualizar en la figura 27.



The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following C++ code:

```
for: FOR controlSymbol VARIABLE FROM exp TO exp STEP exp DO stmtlist END_FOR
{
    //Create a new for statement node
    $$ = new lp::ForStmt($3 , $5, $7, $11, $9);

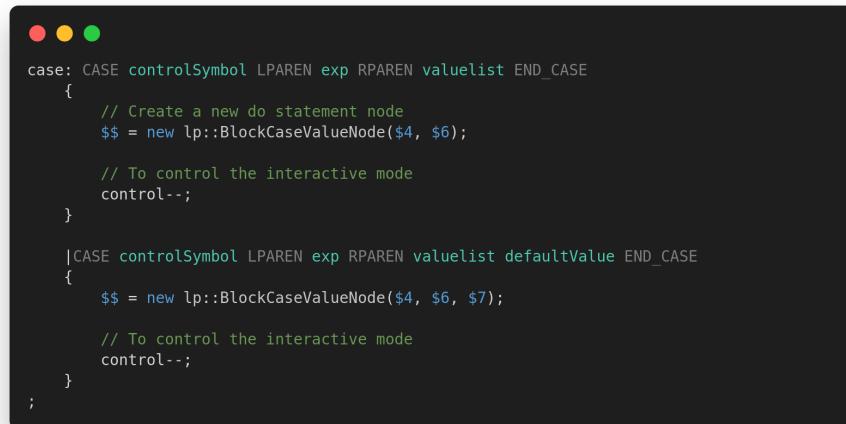
    //TO control the interactive mode
    control--;
}
|
FOR controlSymbol VARIABLE FROM exp TO exp DO stmtlist END_FOR
{
    //Create a new for statement node
    $$ = new lp::ForStmt($3 , $5, $7, $9);

    //TO control the interactive mode
    control--;
}
;
```

Figura 27: Acción semántica símbolo for

#### 5.4.10. case

La acción semántica del símbolo terminal **case** se puede visualizar en la figura 28.



```
case: CASE controlSymbol LPAREN exp RPAREN valuelist END_CASE
{
    // Create a new do statement node
    $$ = new lp::BlockCaseValueNode($4, $6);

    // To control the interactive mode
    control--;
}

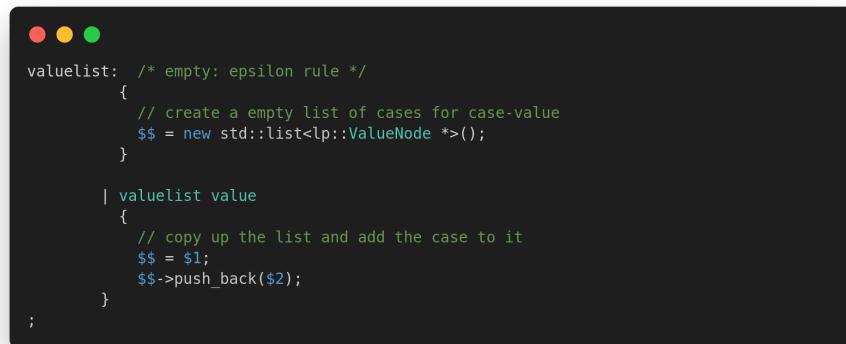
|CASE controlSymbol LPAREN exp RPAREN valuelist defaultValue END_CASE
{
    $$ = new lp::BlockCaseValueNode($4, $6, $7);

    // To control the interactive mode
    control--;
}
;
```

Figura 28: Acción semántica símbolo case

#### 5.4.11. valueList

La acción semántica del símbolo no terminal **valueList** se puede visualizar en la figura 29.



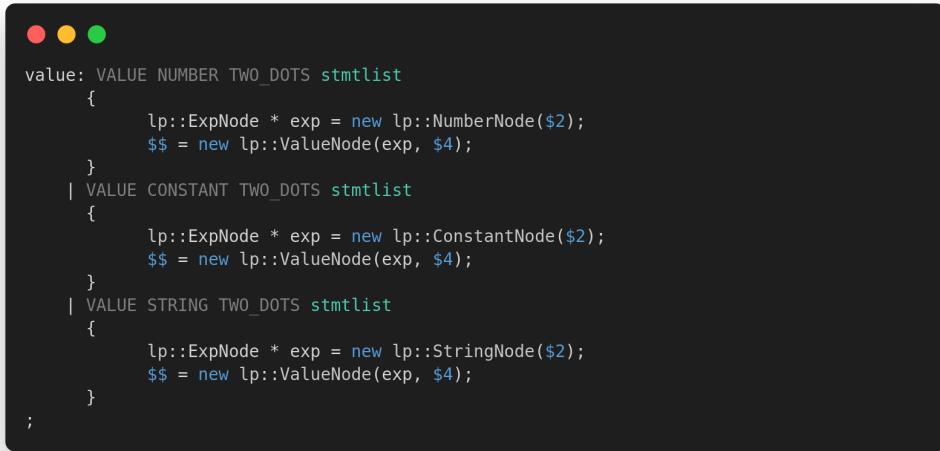
```
valuelist: /* empty: epsilon rule */
{
    // create a empty list of cases for case-value
    $$ = new std::list<lp::ValueNode *>();
}

| valuelist value
{
    // copy up the list and add the case to it
    $$ = $1;
    $$->push_back($2);
}
;
```

Figura 29: Acción semántica símbolo valueList

#### 5.4.12. value

La acción semántica del símbolo terminal **value** se puede visualizar en la figura 30.



```
value: VALUE NUMBER TWO_DOTS stmtlist
{
    lp::ExpNode * exp = new lp::NumberNode($2);
    $$ = new lp::ValueNode(exp, $4);
}
| VALUE CONSTANT TWO_DOTS stmtlist
{
    lp::ExpNode * exp = new lp::ConstantNode($2);
    $$ = new lp::ValueNode(exp, $4);
}
| VALUE STRING TWO_DOTS stmtlist
{
    lp::ExpNode * exp = new lp::StringNode($2);
    $$ = new lp::ValueNode(exp, $4);
}
;
```

Figura 30: Acción semántica símbolo value

#### 5.4.13. cond

La acción semántica del símbolo no terminal **cond** se puede visualizar en la figura 31.

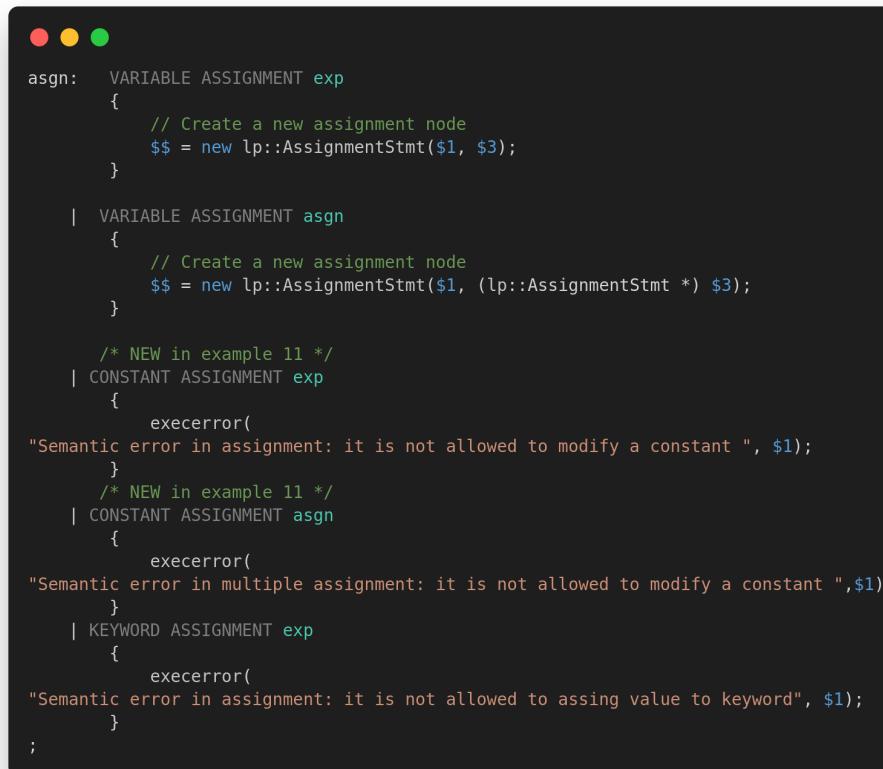


```
cond: LPAREN exp RPAREN
{
    $$ = $2;
}
;
```

Figura 31: Acción semántica símbolo cond

#### 5.4.14. asgn

La acción semántica del símbolo no terminal **asgn** se puede visualizar en la figura 32.



```
asgn: VARIABLE ASSIGNMENT exp
{
    // Create a new assignment node
    $$ = new lp::AssignmentStmt($1, $3);
}

| VARIABLE ASSIGNMENT asgn
{
    // Create a new assignment node
    $$ = new lp::AssignmentStmt($1, (lp::AssignmentStmt *) $3);
}

/* NEW in example 11 */
| CONSTANT ASSIGNMENT exp
{
    execerror(
"Semantic error in assignment: it is not allowed to modify a constant ", $1);
}
/* NEW in example 11 */
| CONSTANT ASSIGNMENT asgn
{
    execerror(
"Semantic error in multiple assignment: it is not allowed to modify a constant ",$1);
}
| KEYWORD ASSIGNMENT exp
{
    execerror(
"Semantic error in assignment: it is not allowed to assing value to keyword", $1);
}
;
```

Figura 32: Acción semántica símbolo asgn

#### 5.4.15. clear\_screen

La acción semántica del símbolo no terminal **clear\_screen** se puede visualizar en la figura 33.

```
● ● ●
clear_screen: CLEAR_SCREEN_TOKEN SEMICOLON
{
    // Create a new clear screen node
    $$ = new lp::ClearScreenStmt();
}
;
```

Figura 33: Acción semántica símbolo clear\_screen

#### 5.4.16. place

La acción semántica del símbolo no terminal **place** se puede visualizar en la figura 34.

```
● ● ●
place: PLACE_TOKEN LPAREN exp COMMA exp RPAREN SEMICOLON
{
    // Create a new place node
    $$ = new lp::PlaceStmt($3, $5);
}
;
```

Figura 34: Acción semántica símbolo place

#### 5.4.17. print

La acción semántica del símbolo terminal **print** se puede visualizar en la figura 35.

```
● ● ●  
print: PRINT exp  
{  
    // Create a new print node  
    $$ = new lp::PrintStmt($2);  
}  
;
```

Figura 35: Acción semántica símbolo print

#### 5.4.18. print\_string

La acción semántica del símbolo terminal **print\_string** se puede visualizar en la figura 36.

```
● ● ●  
print_string: PRINT_STRING exp  
{  
    // Create a new print string node  
    $$ = new lp::PrintStringStmt($2);  
}  
;
```

Figura 36: Acción semántica símbolo print\_string

#### 5.4.19. read

La acción semántica del símbolo terminal **read** se puede visualizar en la figura 37.

```
● ● ●
read: READ LPAREN VARIABLE RPAREN
{
    // Create a new read node
    $$ = new lp::ReadStmt($3);
}

/* NEW rule in example 11 */
| READ LPAREN CONSTANT RPAREN
{
    execerror(
        "Semantic error in \"read statement\": it is not allowed to modify a constant ",$3);
}
;
```

Figura 37: Acción semántica símbolo read

#### 5.4.20. read\_string

La acción semántica del símbolo terminal **read\_string** se puede visualizar en la figura 38.

```
● ● ●
read_string: READ_STRING LPAREN VARIABLE RPAREN
{
    // Create a new read string node
    $$ = new lp::ReadStringStmt($3);
}
;
```

Figura 38: Acción semántica símbolo read\_string

#### 5.4.21. repeat

La acción semántica del símbolo terminal **repeat** se puede visualizar en la figura 39.

```
repeat : REPEAT controlSymbol stmtlist UNTIL cond SEMICOLON
{
    //Create a new do statement node
    $$ = new lp::RepeatStmt($3,$5);

    //To control the interactive mode
    control--;
}
```

Figura 39: Acción semántica símbolo repeat

#### 5.4.22. exp

La acción semántica del símbolo no terminal **block** se puede visualizar en las figuras 40, 41, 42, 43, 44, 45.

```
exp:    NUMBER
{
    // Create a new number node
    $$ = new lp::NumberNode($1);
}
| STRING
{
    // Create a new string node
    $$ = new lp::StringNode($1);
}
| exp PLUS exp
{
    // Create a new plus node
    $$ = new lp::PlusNode($1, $3);
}
| exp MINUS exp
{
    // Create a new minus node
    $$ = new lp::MinusNode($1, $3);
}
| exp MULTIPLICATION exp
{
    // Create a new multiplication node
    $$ = new lp::MultiplicationNode($1, $3);
}
| exp DIVISION exp
{
    // Create a new division node
    $$ = new lp::DivisionNode($1, $3);
}
```

Figura 40: Acción semántica símbolo exp I

```
| exp INT_DIVISION exp
| {
|   // Create a new integer division node
|   $$ = new lp::IntegerDivisionNode($1, $3);
| }

| LPAREN exp RPAREN
| {
|   // just copy up the expression node
|   $$ = $2;
| }

| PLUS exp %prec UNARY
| {
|   // Create a new unary plus node
|   $$ = new lp::UnaryPlusNode($2);
| }

| MINUS exp %prec UNARY
| {
|   // Create a new unary minus node
|   $$ = new lp::UnaryMinusNode($2);
| }

| exp MODULO exp
| {
|   // Create a new modulo node
|
|   $$ = new lp::ModuloNode($1, $3);
| }

| exp POWER exp
| {
|   // Create a new power node
|   $$ = new lp::PowerNode($1, $3);
| }
```

Figura 41: Acción semántica símbolo exp II



```
| VARIABLE
| {
|     // Create a new variable node
|     $$ = new lp::VariableNode($1);
| }
|
| CONSTANT
| {
|     // Create a new constant node
|     $$ = new lp::ConstantNode($1);
| }
```

Figura 42: Acción semántica símbolo exp III



```
| BUILTIN LPAREN listOfExp RPAREN
{
    // Get the identifier in the table of symbols as Builtin
    lp::Builtin *f= (lp::Builtin *) table.getSymbol($1);

    // Check the number of parameters
    if (f->getNParameters() == (int) $3->size())
    {
        switch(f->getNParameters())
        {
            case 0:
            {
                // Create a new Builtin Function with 0 parameters node
                $$ = new lp::BuiltinFunctionNode_0($1);
            }
            break;

            case 1:
            {
                // Get the expression from the list of expressions
                lp::ExpNode *e = $3->front();

                // Create a new Builtin Function with 1 parameter node
                $$ = new lp::BuiltinFunctionNode_1($1,e);
            }
            break;

            case 2:
            {
                // Get the expressions from the list of expressions
                lp::ExpNode *e1 = $3->front();
                $3->pop_front();
                lp::ExpNode *e2 = $3->front();

                // Create a new Builtin Function with 2 parameters node
                $$ = new lp::BuiltinFunctionNode_2($1,e1,e2);
            }
            break;

            default:
                execerror("Syntax error: too many parameters for function ", $1);
        }
    }
    else
        execerror("Syntax error: incompatible number of parameters for function", $1);
}
```

Figura 43: Acción semántica símbolo exp IV

```
| exp GREATER_THAN exp
{
    // Create a new "greater than" node
    $$ = new lp::GreaterThanNode($1,$3);
}

| exp GREATER_OR_EQUAL exp
{
    // Create a new "greater or equal" node
    $$ = new lp::GreaterOrEqualNode($1,$3);
}

| exp LESS_THAN exp
{
    // Create a new "less than" node
    $$ = new lp::LessThanNode($1,$3);
}

| exp LESS_OR_EQUAL exp
{
    // Create a new "less or equal" node
    $$ = new lp::LessOrEqualNode($1,$3);
}

| exp EQUAL exp
{
    // Create a new "equal" node
    $$ = new lp::EqualNode($1,$3);
}

| exp NOT_EQUAL exp
{
    // Create a new "not equal" node
    $$ = new lp::NotEqualNode($1,$3);
}
```

Figura 44: Acción semántica símbolo exp V

```

| exp AND exp
{
    // Create a new "logic and" node
    $$ = new lp::AndNode($1,$3);
}

| exp CONCATENATION exp
{
    $$ = new lp::ConcatenationNode($1,$3);
}

| exp OR exp
{
    // Create a new "logic or" node
    $$ = new lp::OrNode($1,$3);
}

| NOT exp
{
    // Create a new "logic negation" node
    $$ = new lp::NotNode($2);
}
;

```

Figura 45: Acción semántica símbolo exp VI

#### 5.4.23. `listOfExp`

La acción semántica del símbolo no terminal `listOfExp` se puede visualizar en la figura 46.

```

listOfExp:
/* Empty list of numeric expressions */
{
    // Create a new list STL
    $$ = new std::list<lp::ExpNode *>();
}

| exp restOfListOfExp
{
    $$ = $2;

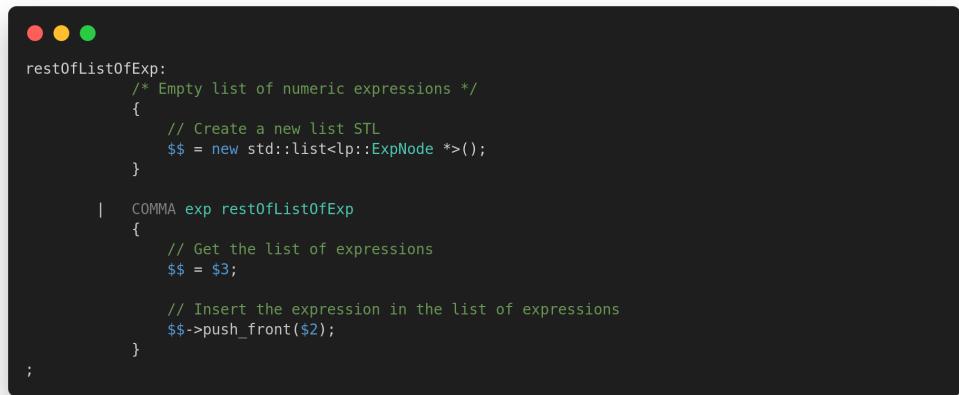
    // Insert the expression in the list of expressions
    $$->push_front($1);
}
;

```

Figura 46: Acción semántica símbolo `listOfExp`

#### 5.4.24. restListOfExp

La acción semántica del símbolo no terminal **restListOfExp** se puede visualizar en la figura 47.



```
restListOfExp:
    /* Empty list of numeric expressions */
    {
        // Create a new list STL
        $$ = new std::list<lp::ExpNode *>();
    }

    | COMMA exp restListOfExp
    {
        // Get the list of expressions
        $$ = $3;

        // Insert the expression in the list of expressions
        $$->push_front($2);
    }
;
```

Figura 47: Acción semántica símbolo restListOfExp

## 6. Código de AST

El código AST (Abstract Syntax Tree) es la parte más compleja del proyecto, que consta de distintas subclases, que son las bases para todas las acciones del intérprete.

### 6.1. Clase Statement

La clase Statement representa las propias declaraciones del lenguaje. Es usada por todas las clases que requieran de realizar asignaciones u operaciones iterativas. Heredando de la clase Statement, tenemos diferentes clases, representadas en la figura 48.

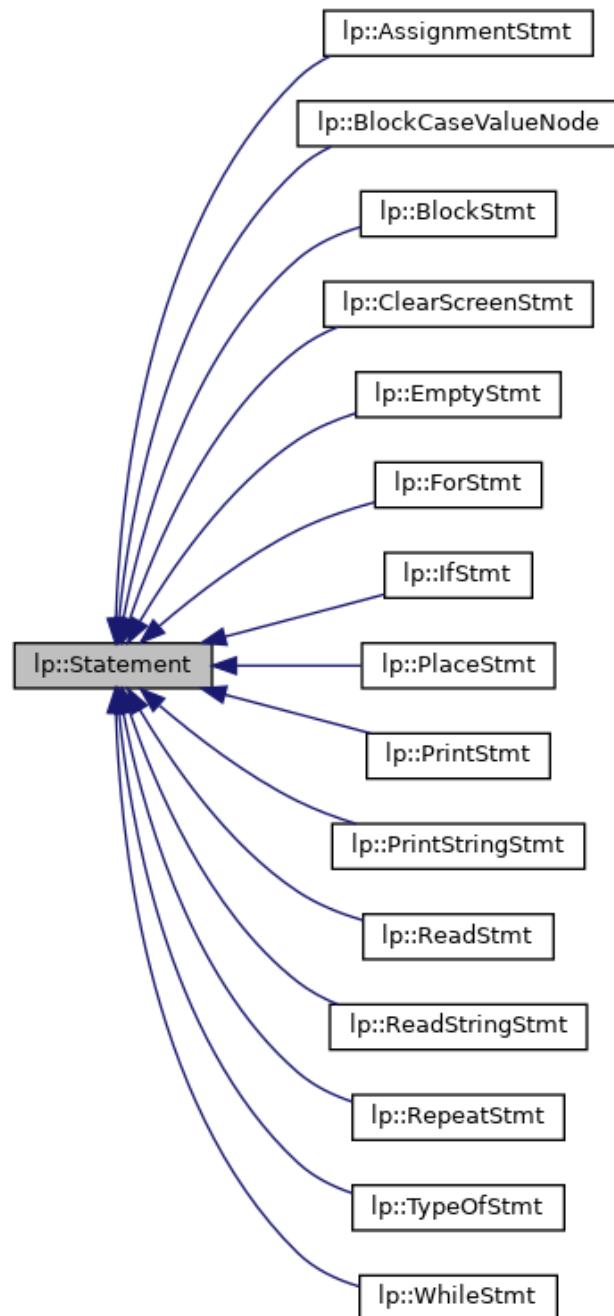


Figura 48: Representación de la clase Statement

### **6.1.1. AssignmentStmt**

Es la declaración correspondiente a las asignaciones. Se encarga de comprobar que el tipo de valor que se va a asignar a la variable sea valido, y gestiona la definición de la variable en la tabla de símbolos, ya sea por primera vez, o una reinstalación en la tabla.

### **6.1.2. BlockCaseValueNode**

Es la base del bucle Case, conteniendo la expresión a evaluar, una lista con los casos a evaluar y el caso por defecto. La evaluación consiste en evaluar los posibles casos para usar la función de evaluación que corresponda, según el tipo de expresión.

### **6.1.3. ClearScreenStmt**

Únicamente se encarga de limpiar la terminal por pantalla usando una macro.

### **6.1.4. EmptyStmt**

No almacena ningún valor ni realiza ninguna acción, representa un valor nulo o vacío. Necesario para la correcta ejecución de ciertas funciones.

### **6.1.5. ForStmt**

Es la base del bucle For, conteniendo la variable sobre la que se itera, la expresión de inicio y la final, y el paso que se da en cada iteración. Además, contiene un *blockStmt*, un conjunto de declaraciones a realizar en cada iteración. La evaluación consiste en una serie de comprobaciones para evitar errores como bucles infinitos o crear la variable sobre la que iterar con un valor predeterminado en caso de no existir. Su funcionamiento es un bucle for propio del lenguaje C.

### **6.1.6. IfStmt**

Es la base del bucle condicional If, conteniendo la condición a evaluar, las acciones consecuentes y las alternativas en caso de existir. La evaluación consiste en la ejecución de un bucle If propio del lenguaje C.

### **6.1.7. PlaceStmt**

Contiene las dos coordenadas para el eje X e Y respectivamente. Su evaluación consiste en la activación de una macro para situar el cursor de escritura en la posición específica de la terminal.

### **6.1.8. PrintStmt**

Contiene la expresión a imprimir. Su evaluación consiste en la comprobación del tipo, ya que debe ser numérico o booleano.

### **6.1.9. PrintStringStmt**

Contiene la expresión a imprimir. Su evaluación consiste en la comprobación del tipo, ya que debe ser únicamente una cadena de texto.

### **6.1.10. ReadStmt**

Contiene el nombre de la variable donde se almacenará el valor a leer por pantalla. La evaluación consiste en asignar a la variable indicada como argumento el valor leído por pantalla, comprobando que su tipo sea numérico o booleano.

### **6.1.11. ReadStringStmt**

Contiene el nombre de la variable donde se almacenará el valor a leer por pantalla. La evaluación consiste en asignar a la variable indicada como argumento el valor leído por pantalla, comprobando que su tipo sea una cadena de texto.

### **6.1.12. RepeatStmt**

Es la base del bucle Repeat. Contiene la expresión que se evalúa como condición de iteración y la lista de sentencias a ejecutar en cada iteración. Su evaluación consiste en la ejecución del bucle While propio de C, pero negando la condición, es decir, ejecutando mientras no se cumpla la condición negada.

### **6.1.13. TypeOfStmt**

Contiene la expresión cuyo tipo queremos saber. Su evaluación consiste en la ejecución de un bucle Switch propio de C, que devuelve un mensaje u otro según el tipo de la variable.

### **6.1.14. WhileStmt**

Es la base del bucle While. Contiene la expresión que se evalúa como condición de iteración y la lista de sentencias a ejecutar. La evaluación consiste en la ejecución de un bucle While propio de C, iterando mientras se cumpla la condición evaluada.

### **6.1.15. BlockStmt**

Almacena un bloque o lista de declaraciones, es decir, un conjunto de cualquiera de las previamente mencionadas declaraciones. Al evaluarse, se evalúan todos los elementos de la lista por separado.

## 6.2. Clase ExpNode

La clase ExpNode representa a las expresiones, propias de las variables y la relación entre estas. Gran parte del código ya estaba escrito previamente, por lo que nos hemos limitado a cambios menores, como añadir una nueva operación numérica y el tipo de dato Cadena.

De la clase ExpNode heredan todas las clases que serán nombradas a continuación. Solo se entrarán en detalles sobre las clases nuevas o modificadas en profundidad.

La estructura jerárquica de la clase ExpNode se puede visualizar en la figura 49.

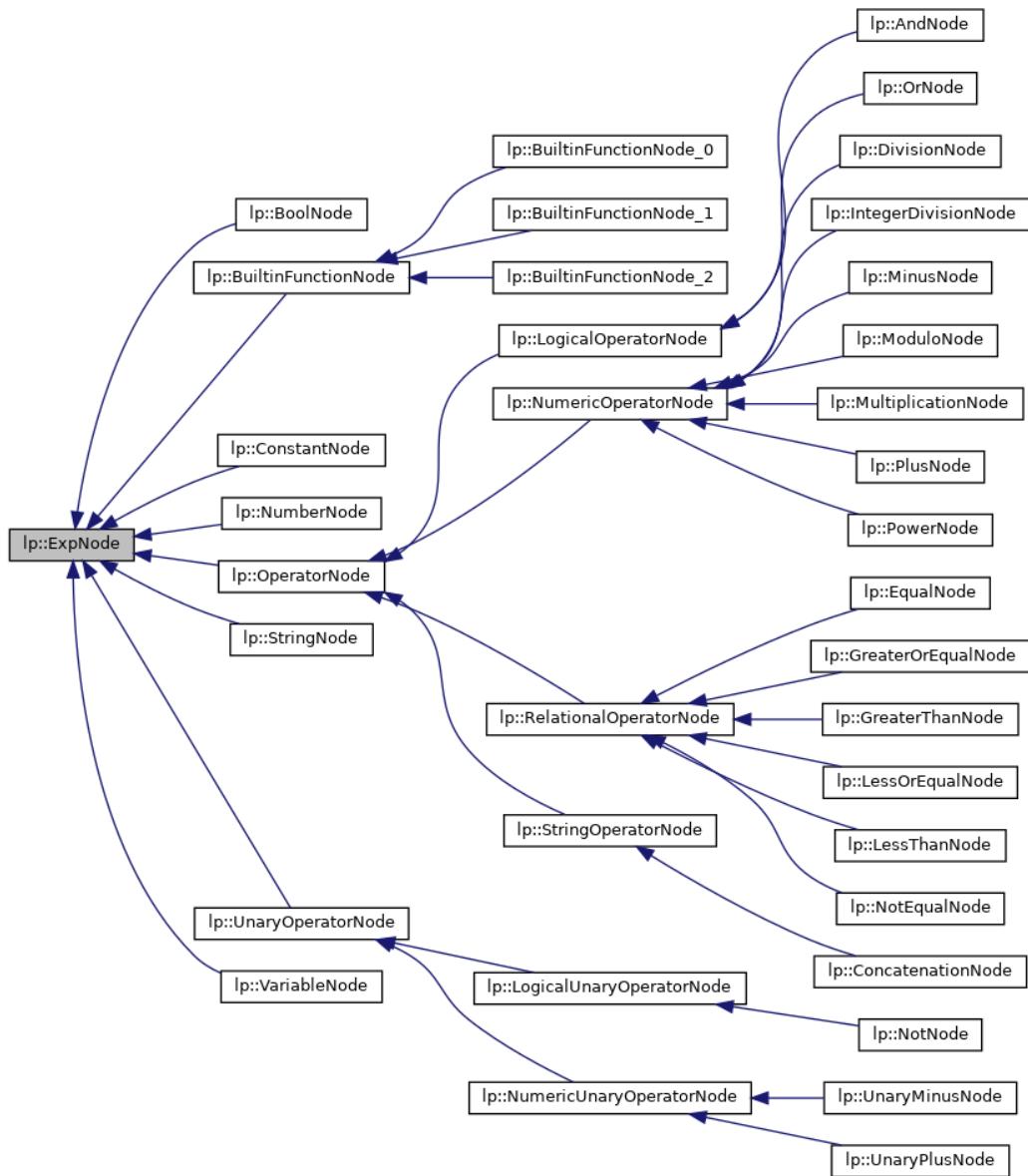


Figura 49: Representación de la clase ExpNode

Las clases ya creadas previamente y no modificadas han sido:

1. VariableNode
2. NumberNode
3. BoolNode
4. BuiltinFunctionNode
5. OperatorNode
6. PlusNode
7. MinusNode
8. UnaryPlusNode
9. UnaryMinusNode
10. MultiplicationNode
11. DivisionNode
12. ModuloNode
13. PowerNode
14. AndNode
15. OrNode
16. NotNode
17. EqualNode
18. NotEqualNode
19. GreaterThanNode
20. GreaterOrEqualNode
21. LessThanNode
22. LessOrEqualThanNode

Los principales añadidos a la clase ExpNode han sido:

### 6.2.1. **StringNode**

Esta clase contiene un valor del tipo String, propio del lenguaje C. Su única acción consiste en devolver el valor de la cadena.

### 6.2.2. **ConcatenationNode**

Esta clase contiene dos StringNode en forma de ExpNodes. Su evaluación consiste en comprobar que ambos sean del tipo cadena de texto, para posteriormente concatenarlos y devolver la cadena ya concatenada. Hereda de un nuevo tipo de ExpNode, el **StringOperatorNode**, un nuevo tipo de **OperatorNode** que almacena dos valores del tipo String para poder operar con ellos.

### 6.2.3. **IntegerDivisionNode**

Esta clase contiene los dos valores numéricos enteros con los que operar. Su evaluación consiste en comprobar que no se realiza una división por cero, y devolver el valor la división entera.

## 7. Funciones auxiliares

En cuanto a funciones auxiliares solo tenemos una auxiliar, la función **Type\_of()**, que recibe como argumento una variable e indica el tipo de valor que esta almacena, ya sea una cadena, un numero, un valor booleano o algo indefinido. Es de gran utilidad para la comprobación de errores, y es una parte crucial del funcionamiento del bucle Case.

## 8. Modo de obtención del intérprete

En cuanto al modo de obtención del intérprete, daremos una descripción del sistema de directorios y de los ficheros que componen el proyecto en su totalidad.

Teniendo en cuenta que todos los directorios parten de un mismo directorio raíz, los directorios son:

- **ast:** Contiene el código necesario para la ejecución de la clase AST.
- **error:** Contiene el código necesario para la detección y tratamiento de errores.
- **includes:** Contiene un fichero de cabecera auxiliar para el uso de macros visuales.
- **parser:** Contiene todo el código respectivo al analizador léxico y al analizador sintáctico.
- **table:** Contiene todo el código respectivo a la tabla de símbolos.
- **examples:** No es propiamente un directorio del proyecto, contiene archivos de texto a modo de ejemplo, para poder probar el funcionamiento del interprete.
- **html:** Tampoco es un directorio propio del proyecto, contiene los archivos correspondientes a la documentación generada por Doxygen.

Respecto a los archivos, podemos centrarnos en cada directorio.

### 8.1. Directorio /ast

- **ast.cpp:** Implementación de las clases y funciones del AST.
- **ast.hpp:** Declaración de las clases y funciones del AST.
- **makefile:** Crea código ejecutable que luego serán usados en una compilación general.

## 8.2. Directorio /error

Contiene las funciones que controlan y notifican errores detectados durante la interpretación del código.

- **error.hpp:** Implementación de las funciones de error.
- **error.cpp:** Archivo de cabeceras para las funciones de error.
- **makefile:** Crea código ejecutable que luego serán usados en una compilación general.

## 8.3. Directorio /includes

Contiene un solo archivo, *macros.hpp*, que a su vez contiene una serie de macros, orientadas a la visualización del texto, tanto en el interprete como en los códigos interpretados.

## 8.4. Directorio /parser

Contiene los ficheros usados para el análisis léxico y el análisis sintáctico, escritos en flex y yacc. Además, contiene un makefile que genera el código ejecutable, que luego será usado en una compilación general.

## 8.5. Directorio /table

Contiene los archivos referentes a las clases de los tipos de datos que se usan en el intérprete de datos a utilizar en el intérprete, así como de la creación y gestión de la tabla de símbolos.

## 8.6. Directorio /examples

El directorio examples solo contiene ficheros de texto con programas de ejemplo para ser interpretados por el interprete, ejecutándose desde el fichero en lugar de en modo interactivo.

## 8.7. Directorio /html

Contiene toda la documentación generada usando Doxygen.

## 8.8. Directorio raíz

Además, podemos mencionar los archivos propios del directorio raíz:

- interpreter.cpp: Programa principal del intérprete.
- Doxyfile: Archivo de creación de la documentación en Doxygen.
- makefile: Genera el código ejecutable en el archivo *interpreter.exe*

Finalmente, el diagrama que representa esa jerarquía de directorios del proyecto sería 50

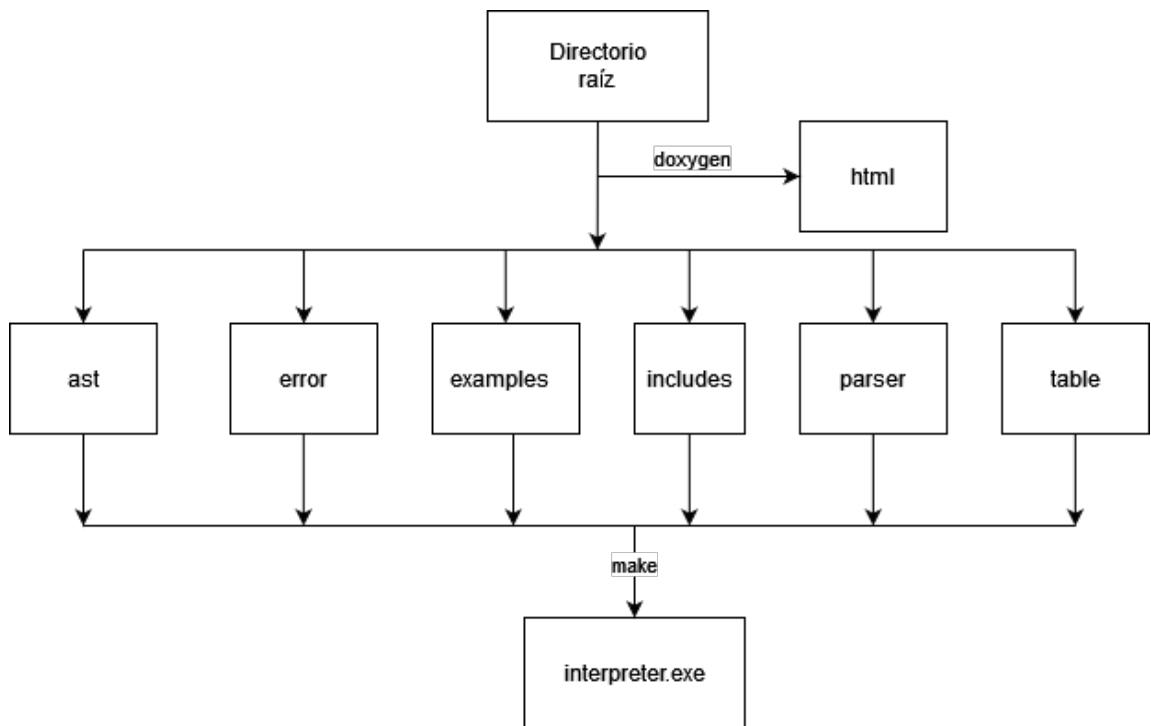


Figura 50: Representación de la jerarquía de ficheros

## **9. Modo de ejecución del intérprete**

El intérprete de pseudocódigo tiene dos modos de ejecución, un modo interactivo y una ejecución de lectura de un fichero de texto.

### **9.1. Modo interactivo**

En el modo interactivo, el programa queda a la espera de instrucciones introducidas por el usuario en linea de comandos. Irá ejecutando todas las instrucciones en tiempo de ejecución. La forma de ejecución, para tener un acceso libre e interactivo al intérprete en modo interactivo sería:

```
./interpreter.exe
```

### **9.2. Ejecución desde un fichero**

En el modo de ejecución desde un fichero, el intérprete ejecuta las instrucciones escritas en un archivo de texto, pasado al programa como argumento. El usuario no tiene interacción directa con el programa, salvo pasarle el archivo al programa como argumento. La forma de ejecución, indicando el fichero como argumento, sería:

```
./ interpreter archivo.p
```

## 10. Ejemplos

En cuanto a ejemplos ejecutables podemos diferenciar dos tipos, los ya dados como ejemplos y los hechos por nosotros mismos.

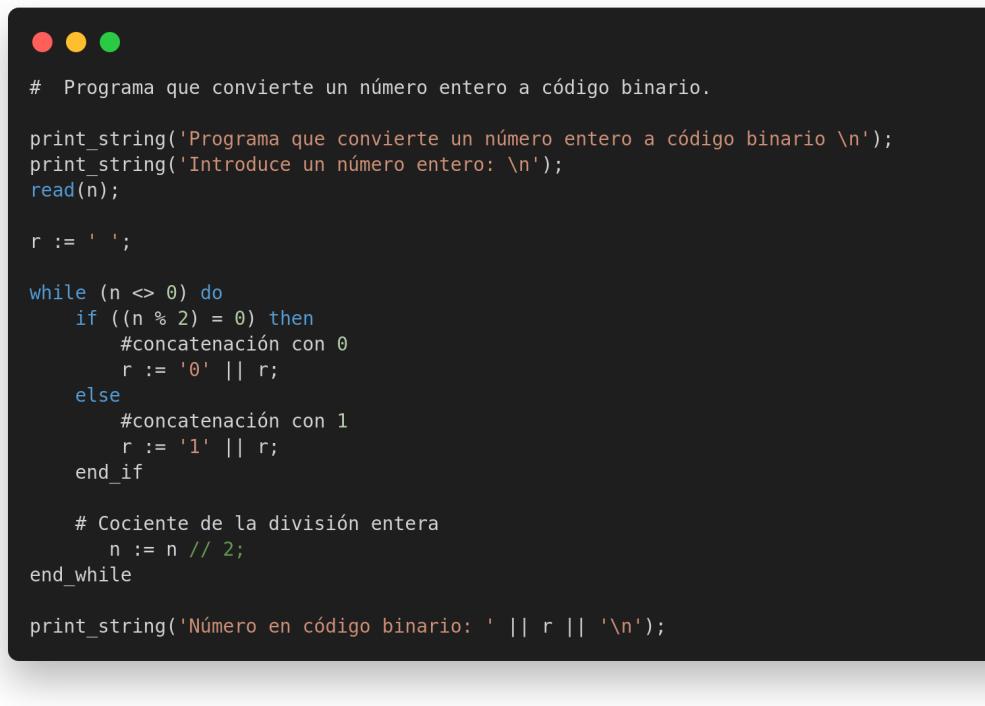
### 10.1. Ejemplos precreados

#### 10.1.1. binario.p

Ejemplo diseñado para mostrar el correcto funcionamiento de los bucles if, while, concatenación de cadenas, comentarios y divisiones enteras. Consiste en traducir un número entero a su notación en código binario. La instrucción para la ejecución sería:

```
./interpreter.exe examples/binario.p
```

El código del ejemplo se puede visualizar en la figura 51.



```
# Programa que convierte un número entero a código binario.

print_string('Programa que convierte un número entero a código binario \n');
print_string('Introduce un número entero: \n');
read(n);

r := ' ';

while (n <> 0) do
    if ((n % 2) = 0) then
        #concatenación con 0
        r := '0' || r;
    else
        #concatenación con 1
        r := '1' || r;
    end_if

    # Cociente de la división entera
    n := n // 2;
end_while

print_string('Número en código binario: ' || r || '\n');
```

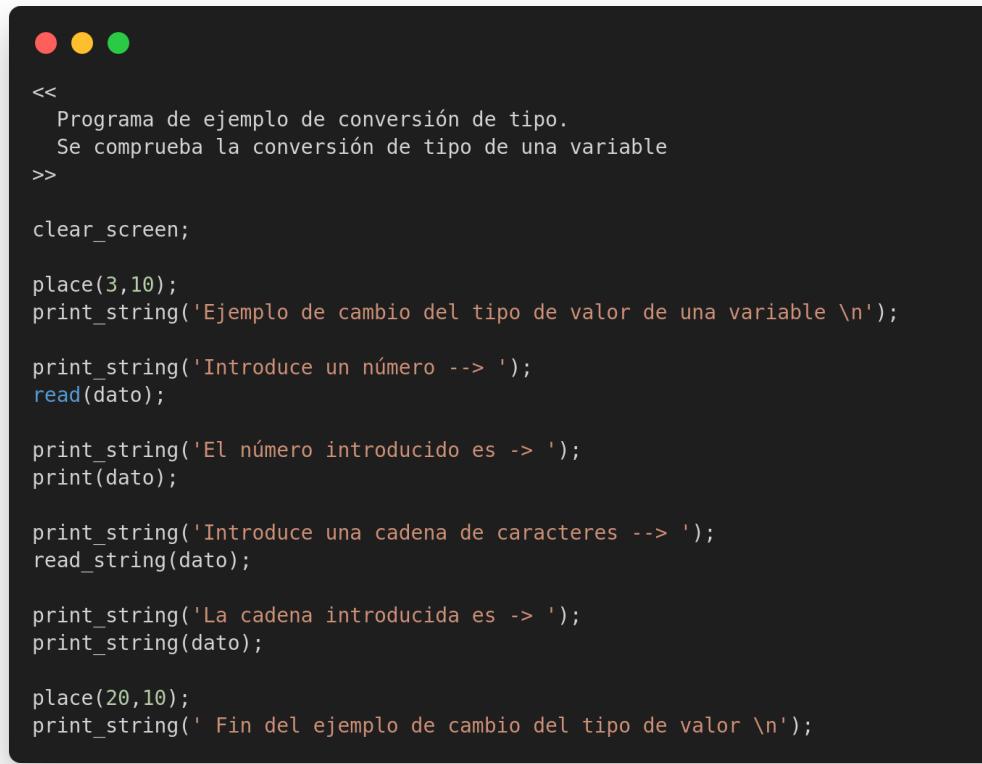
Figura 51: Código del ejemplo binario.p

### 10.1.2. conversion.p

Ejemplo diseñado para mostrar el correcto funcionamiento de la lectura y escritura de variables, tanto numéricas como alfanuméricas. Consiste en leer y devolver por pantalla un valor numérico y otro alfanumérico. La instrucción para la ejecución sería:

```
./interpreter.exe examples/conversion.p
```

El código del ejemplo se puede visualizar en la figura 52.



```
<<
    Programa de ejemplo de conversión de tipo.
    Se comprueba la conversión de tipo de una variable
>>

clear_screen;

place(3,10);
print_string('Ejemplo de cambio del tipo de valor de una variable \n');

print_string('Introduce un número --> ');
read(data);

print_string('El número introducido es -> ');
print(data);

print_string('Introduce una cadena de caracteres --> ');
read_string(data);

print_string('La cadena introducida es -> ');
print_string(data);

place(20,10);
print_string(' Fin del ejemplo de cambio del tipo de valor \n');
```

Figura 52: Código del ejemplo conversion.p

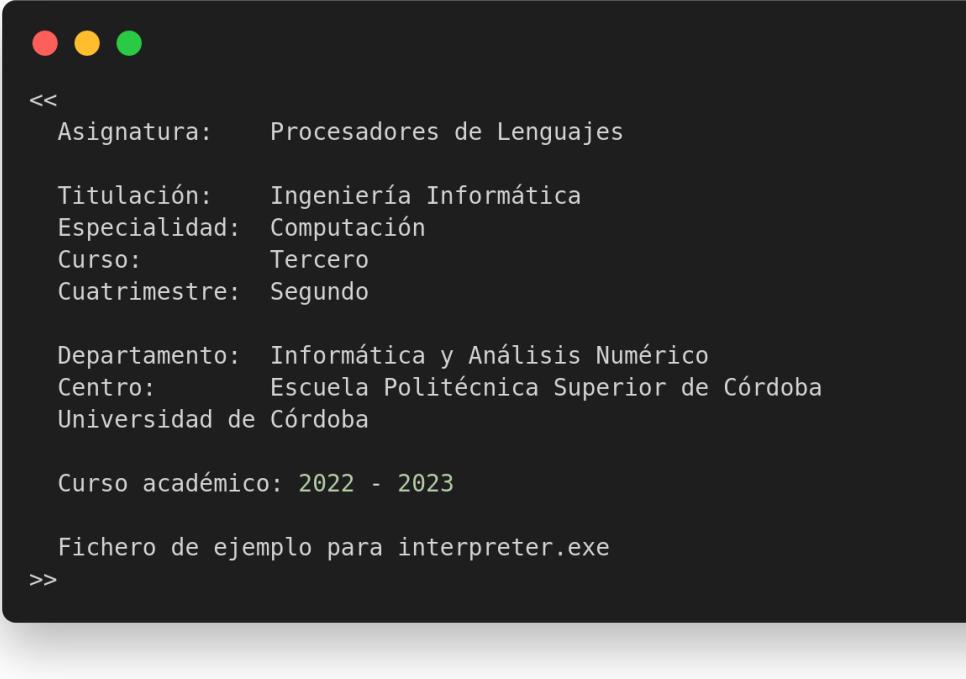
### 10.1.3. menu.p

Este ejemplo nos permite introducir nuestro nombre, para después acceder a un menú que nos permite calcular el factorial de un numero, el máximo común divisor entre dos números enteros, o bien salir del programa.

La instrucción para la ejecución sería:

```
./interpreter.exe examples/menu.p
```

El código del ejemplo se puede visualizar en las figuras 53, 54, 55, 56, 57, 58.



```
Asignatura: Procesadores de Lenguajes
Titulación: Ingeniería Informática
Especialidad: Computación
Curso: Tercero
Cuatrimestre: Segundo

Departamento: Informática y Análisis Numérico
Centro: Escuela Politécnica Superior de Córdoba
Universidad de Córdoba

Curso académico: 2022 - 2023

Fichero de ejemplo para interpreter.exe
>>
```

Figura 53: Código del ejemplo menu.p I

```
# Bienvenida

clear_screen;
place(10,10);
print_string('Introduce tu nombre --> ');
read_string(nombre);

clear_screen;
place(10,10);
print_string(' Bienvenido/a << ');
print_string(nombre);
print_string(' >> a \'interpreter.exe\' .');

place(40,10);
print_string('Pulsa una tecla para continuar');
read_string( pausa);
```

Figura 54: Código del ejemplo menu.p II



```
repeat

# Opciones disponibles

clear_screen;
place(10,10);
print_string(' Factorial de un número --> 1 ');

place(11,10);
print_string(' Máximo común divisor ----> 2 ');

place(12,10);
print_string(' Finalizar -----> 0 ');

place(15,10);
print_string(' Elige una opcion ');

read(opcion);

clear_screen;

# Fin del programa
```

Figura 55: Código del ejemplo menu.p III

```
if (opcion = 0)
    then
        place(10,10);
        print_string(nombre);
        print_string(': gracias por usar el intérprete ipe.exe ');
else
    # Factorial de un número
    if (opcion = 1)
        then
            place(10,10);
            print_string(' Factorial de un numero ');

            place(11,10);
            print_string(' Introduce un numero entero ');
            read(N);

            factorial := 1;

            for i from 2 to N step 1 do
                factorial := factorial * i;
            end_for;

            # Resultado
            place(15,10);
            print_string(' El factorial de ');
            print(N);
            print_string(' es ');
            print(factorial);
```

Figura 56: Código del ejemplo menu.p IV

```
else
    # Máximo común divisor
    if (opcion = 2)
        then
            place(10,10);
            print_string(' Máximo común divisor de dos números ');

            place(11,10);
            print_string(' Algoritmo de Euclides ');

            place(12,10);
            print_string(' Escribe el primer número ');
            read(a);

            place(13,10);
            print_string(' Escribe el segundo número ');
            read(b);

            # Se ordenan los números
            if (a < b)
                then
                    auxiliar := a;
                    a := b;
                    b := auxiliar;
                end_if;

            # Se guardan los valores originales
            A1 := a;
            B1 := b;

            # Se aplica el método de Euclides
            resto := a % b;

            while (resto <> 0) do
                a := b;
                b := resto;
                resto := a % b;
            end_while;

            # Se muestra el resultado
            place(15,10);
            print_string(' Máximo común divisor de ');
            print(A1);
            print_string(' y ');
            print(B1);
            print_string(' es ---> ');
            print(b);
```

Figura 57: Código del ejemplo menu.p V

```
# Resto de opciones
else
    place(15,10);
    print_string(' Opcion incorrecta ');
    end_if;
end_if;
end_if;

place(40,10);
print_string('\n Pulse una tecla para continuar --> ');
read_string(pausa);

until (opcion = 0);

# Despedida final
clear_screen;
place(10,10);
print_string('El programa ha concluido');
```

Figura 58: Código del ejemplo menu.p VI

## 10.2. Ejemplos propios

Hemos realizado dos ejemplos propios, para probar algunos de los elementos que no han sido probados en los ejemplos ya dados.

### 10.3. fibonacci.p

Este ejemplo imita el ejemplo **menu.p**, permitiéndonos calcular la sucesión de fibonacci a partir de un número entero introducido por pantalla. La instrucción para su ejecución sería:

```
./interpreter.exe examples/fibonacci.p
```

El código del ejemplo se puede visualizar en las figuras 59, 60 y 61.

```
<<
Asignatura:    Procesadores de Lenguajes

Titulación:    Ingeniería Informática
Especialidad:  Computación
Curso:         Tercero
Cuatrimestre:  Segundo

Departamento:  Informática y Análisis Numérico
Centro:        Escuela Politécnica Superior de Córdoba
Universidad de Córdoba

Curso académico: 2022 - 2023

Fichero de ejemplo para interpreter.exe
>>

# Bienvenida

clear_screen;
place(10,10);
print_string('Introduce tu nombre -->');
read_string(nombre);

clear_screen;
place(10,10);
print_string(' Bienvenido/a << ');
print_string(nombre);
print_string(' >> a \'interpreter.exe\'.');

place(40,10);
print_string('Pulsa una tecla para continuar');
read_string(pausa);
```

Figura 59: Código del ejemplo fibonnaci.p I



```
repeat

# Opciones disponibles

clear_screen;
place(10,10);
print_string(' Bienvenido al ejemplo Fibonacci');

place(11,10);
print_string(' Escriba un numero');

place(12,10);
print_string(' Finalizar -----> 0');

place(15,10);
print_string(' Escriba un numero, sera su iteracion de la serie de Fibonacci');

read(opcion);

print_string('Entra aqui');

clear_screen;
```

Figura 60: Código del ejemplo fibonnaci.p II

```

● ● ●

a := 0;
b := 1;

for i from 2 to opcion+1 do
    c := a + b;
    a := b;
    b := c;

    print_string('El numero de la iteracion ');
    print(i-1);
    print_string(' es ');
    print(c);

end_for;

place(40,10);
print_string('Pulsa una tecla para continuar');
read_string( pausa);
read_string( pausa);

until (opcion = 0);

# Despedida final
clear_screen;
place(10,10);
print_string('El programa ha concluido');
clear_screen;
place(0,0);

```

Figura 61: Código del ejemplo fibonnaci.p III

## 10.4. `geometrico.p`

Este ejemplo imita el ejemplo `menu.p`, usando una estructura condicional `case` para navegar por las distintas opciones del menú, permitiéndonos calcular áreas geométricas sencillas como el área de un rectángulo, de un triángulo o de un círculo. Para el área del círculo, hemos probado el correcto funcionamiento de las constantes, en este caso, de la constante PI. La instrucción para su ejecución sería:

```
./interpreter.exe examples/geometrico.p
```

El código del ejemplo se puede visualizar en las figuras 62, 63, 64 y 65.

```
<<
    Asignatura:      Procesadores de Lenguajes

    Titulación:     Ingeniería Informática
    Especialidad:   Computación
    Curso:          Tercero
    Cuatrimestre:   Segundo

    Departamento:  Informática y Análisis Numérico
    Centro:        Escuela Politécnica Superior de Córdoba
    Universidad de Córdoba

    Curso académico: 2022 - 2023

    Fichero de ejemplo para interpreter.exe
>>

# Bienvenida

clear_screen;
place(10,10);
print_string('Introduce tu nombre --> ');
read_string(nombre);

clear_screen;
place(10,10);
print_string(' Bienvenido/a << ');
print_string(nombre);
print_string(' >> a \'interpreter.exe\' .');
place(15,10);
print_string('Ejemplo realizado con la estructura Case');

place(40,10);
print_string('Pulsa una tecla para continuar');
read_string( pausa);
```

Figura 62: Código del ejemplo geometrico.p I



```
repeat
    # Opciones disponibles
    clear_screen;
    place(10,10);
    print_string(' Area Rectangulo --> 1 ');
    place(11,10);
    print_string(' Area Circulo ----> 2 ');
    place(12,10);
    print_string(' Area Triangulo --> 3 ');
    place(13,10);
    print_string(' Finalizar -----> 0 ');
    place(16,10);
    print_string(' Elige una opcion ');
    read(opcion);
    clear_screen;
```

Figura 63: Código del ejemplo geometrico.p II

```
case (opcion)
    value 1:
        place(10,10);
        print_string('Introduce la base del rectangulo --> ');
        read(base);
        place(11,10);
        print_string('Introduce la altura del rectangulo --> ');
        read(altura);
        area := base * altura;
        place(12,10);
        print_string('El area del rectangulo es ');
        print(area);
        print_string('\n');

    value 2:
        place(10,10);
        print_string('Introduce el radio del circulo --> ');
        read(radio);
        area := PI * radio * radio;
        place(11,10);
        print_string('El area del circulo es ');
        print(area);
        print_string('\n');

    value 3:
        place(10,10);
        print_string('Introduce la base del triangulo --> ');
        read(base);
        place(11,10);
        print_string('Introduce la altura del triangulo --> ');
        read(altura);
        area := base * altura / 2;
        place(12,10);
        print_string('El area del triangulo es ');
        print(area);
        print_string('\n');

    default:
        place(10,10);
        print_string('Saliendo del programa');
        print_string('\n');
```

Figura 64: Código del ejemplo geometrico.p III

```
end_case;

place(40,10);
print_string('\n Pulse una tecla para continuar --> ');
read_string(pausa);

until (opcion = 0);

# Despedida final
clear_screen;
place(10,10);
print_string('El programa ha concluido');
clear_screen;
place(0,0);
```

Figura 65: Código del ejemplo geometrico.p IV

## 11. Conclusiones

Como conclusión, podemos afirmar que este trabajo práctico nos ha servido para tener una nueva perspectiva respecto a la programación y el desarrollo de código. Después de trabajar con un lenguaje de programación a tan 'bajo nivel', tenemos una mejor comprensión de como funciona el código, las variables, las asignaciones, y las operaciones entre tipos de datos. Con los conocimientos adquiridos, esperamos desarrollar proyectos de programación más eficientes, tanto en lo académico como en lo profesional.

Nuestro intérprete tiene puntos positivos, obviamente. Hemos conseguido implementar todas las características indicadas en la tarea, de forma eficiente y funcional. Hemos realizado una gestión de error de errores eficaz, que detecta errores, los localiza, los describe, y evita la cascada de errores.

Pero lo más notable son los puntos negativos. Aunque hemos conseguido implementar todo lo requerido, nos hemos quedado cortos.

En primer lugar, falta pulido, arreglar error menores que aunque no impiden el correcto funcionamiento del intérprete, pueden resultar en una mala experiencia para el usuario.

En cuanto a características, podríamos haber añadido más operaciones o características, como listas o vectores, escritura en ficheros, o ejemplos mas concretos y originales.

Por último, somos conscientes de fallos y diferencias al ejecutar el intérprete desde un archivo. Aunque al ejecutar el intérprete en modo interactivo funciona todo según lo esperado, desde un fichero nos encontramos con fallos inesperados.

Todo eso se falta por la falta de pulido, y aunque somos capaces de resolver estos problemas, estamos sujetos a la falta de tiempo.

## 12. Bibliografía

[1] C. Donnelly, Bison The Yacc-compatible Parser Generator [en línea], [Consulta: 4 de junio de 2023]. Texto en formato PDF.

Disponible en: <https://www.gnu.org/software/bison/manual/bison.pdf>

[2] A. Aaby, Compiler Construction using Flex and Bison, 204 [en línea], [Consulta: 4 de junio de 2023]. Texto en formato PDF.

Disponible en: [https://moodle.uco.es/m2223/pluginfile.php/327070/mod\\_resource/content/1/2004-Aaby-Flex-Bison.pdf](https://moodle.uco.es/m2223/pluginfile.php/327070/mod_resource/content/1/2004-Aaby-Flex-Bison.pdf)

[3] Prof. Dr. N.L. Fernández García, Guión de prácticas de bison [en línea], [Consulta: 10 de mayo de 2023]. Texto en formato PDF.

Disponible en: [https://moodle.uco.es/m2223/pluginfile.php/46428/mod\\_resource/content/3/Guion-practicas.pdf](https://moodle.uco.es/m2223/pluginfile.php/46428/mod_resource/content/3/Guion-practicas.pdf)

[4] S. Gálvez Rojas, LEX: El Analizador Léxico, [en línea], [Consulta: 10 de mayo de 2023]. Texto en formato PDF.

Disponible en: <http://www.lcc.uma.es/galvez/ftp/tci/TutorialLex.pdf>