

UNIVERSIDAD DE CÓRDOBA

ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA

GRADO DE INGENIERÍA INFORMÁTICA - MENCIÓN EN COMPUTACIÓN

TERCER CURSO - SEGUNDO CUATRIMESTRE - 2020/2021

PROCESADORES DE LENGUAJES

---

## Intérprete de pseudocódigo en español: IPE

---

***Profesor:** Nicolás Luis Fernández García*

***Autores:** Ventura Lucena Martínez, Francisco  
David Castejón Soto*



UNIVERSIDAD DE CÓRDOBA

ESCUELA POLITÉCNICA  
SUPERIOR DE CÓRDOBA  
Universidad de Córdoba



Córdoba, 1 de junio de 2021

## Índice

<b>1</b>	<b>Introducción</b>	<b>4</b>
1.1	Descripción del trabajo . . . . .	4
1.2	Partes del documento . . . . .	4
<b>2</b>	<b>Lenguaje de pseudocódigo</b>	<b>5</b>
2.1	Componentes léxicos . . . . .	5
2.2	Sentencias . . . . .	7
<b>3</b>	<b>Tabla de símbolos</b>	<b>10</b>
3.1	Descripción . . . . .	10
<b>4</b>	<b>Análisis léxico</b>	<b>12</b>
4.1	Descripción . . . . .	12
4.1.1	Área de declaraciones . . . . .	12
4.1.2	Área de reglas . . . . .	13
<b>5</b>	<b>Análisis sintáctico</b>	<b>18</b>
5.1	Símbolos terminales . . . . .	18
5.2	Símbolos no terminales . . . . .	19
5.3	Reglas de producción . . . . .	20
5.3.1	Control de errores . . . . .	23
5.4	Acciones semánticas . . . . .	24
<b>6</b>	<b>AST</b>	<b>40</b>
6.1	Clase <i>Statement</i> . . . . .	41
6.1.1	<i>AssignmentStmt</i> . . . . .	42
6.1.2	<i>CaseStmt</i> . . . . .	43
6.1.3	<i>ClearStmt</i> . . . . .	45
6.1.4	<i>ForStmt</i> . . . . .	46
6.1.5	<i>IfStmt</i> . . . . .	49
6.1.6	<i>MinusEqualStmt</i> . . . . .	52
6.1.7	<i>MinusMinusNode</i> . . . . .	54
6.1.8	<i>PlaceStmt</i> . . . . .	55
6.1.9	<i>PlusEqualStmt</i> . . . . .	57
6.1.10	<i>PlusPlusNode</i> . . . . .	59
6.1.11	<i>ReadStringStmt</i> . . . . .	61
6.1.12	<i>RepeatStmt</i> . . . . .	63
6.1.13	<i>SwitchStmt</i> . . . . .	65
6.1.14	<i>WhileStmt</i> . . . . .	68
6.1.15	<i>WriteStringStmt</i> . . . . .	70
6.2	Clase <i>ExpNode</i> . . . . .	73
6.2.1	<i>DivisionEnteraNode</i> . . . . .	74

6.2.2	<i>ConcatNode</i>	75
6.2.3	<i>StringNode</i>	77
6.2.4	<i>StringOperatorNode</i>	78
<b>7</b>	<b>Funciones auxiliares</b>	<b>80</b>
7.1	Descripción	80
<b>8</b>	<b>Modo de obtención del intérprete</b>	<b>81</b>
<b>9</b>	<b>Modo de ejecución</b>	<b>84</b>
9.1	Interactiva	84
9.2	A partir de un fichero	85
9.2.1	Control de errores	85
<b>10</b>	<b>Ejemplos</b>	<b>86</b>
10.1	menu.e	86
10.2	conversion.e	89
10.3	test_#_y_mayusculas.e	90
10.4	test_booleano.e	91
10.5	test_cadenas.e	92
10.6	test_casos.e	92
10.7	test_error.txt	94
10.8	test_numeros.e	94
10.9	test_operadores_aritmeticos.e	95
10.10	test_operadores_relacionales.e	96
10.11	test_palabras_reservadas.e	97
10.12	test_para.e	100
10.13	test_repetir.e	100
10.14	test_si.e	101
<b>11</b>	<b>Conclusiones</b>	<b>103</b>
11.1	Conclusiones generales	103
11.2	Puntos fuertes y puntos débiles del intérprete	103
11.3	Otras consideraciones	103

## Índice de figuras

1	Jerarquía de clases de <code>SymbolInterface</code> y <code>TableInterface</code> . . . . .	10
2	Jerarquía de herencia de la clase <code>StringVariable</code> . . . . .	11
3	AST - Clase <i>Statement</i> . . . . .	41
4	AST - Clase <i>ExpNode</i> . . . . .	73
5	Ejecución interactiva del intérprete. . . . .	84
6	Ejecución del intérprete desde fichero. . . . .	85

## Listings

1	Área de reglas - Espacios en blanco y tabuladores. . . . .	13
2	Área de reglas - Salto de línea. . . . .	13
3	Área de reglas - Reconocimiento de inicio de cadena. . . . .	13
4	Área de reglas - Reconocimiento de final de cadena. . . . .	14
5	Área de reglas - Reconocimiento de inicio de comentario de varias líneas. . . . .	14
6	Área de reglas - Estado Q1 - Reconocimiento de inicio de comentario de varias líneas. . . . .	14
7	Área de reglas - Estado Q1 - Reconocimiento de final de comentario de varias líneas. . . . .	14
8	Área de reglas - Punto y coma. . . . .	14
9	Área de reglas - Coma. . . . .	14
10	Área de reglas - Valores numéricos. . . . .	15
11	Área de reglas - Identificadores. . . . .	15
12	Área de reglas - Comentarios de una línea. . . . .	15
13	Área de reglas - Operadores y otros. . . . .	16
14	Área de reglas de control de errores - Valores numéricos no válidos. . . . .	16
15	Área de reglas de control de errores - Identificadores no válidos. . . . .	16
16	Área de reglas de control de errores - Otros. . . . .	17
17	Acciones semánticas - program . . . . .	24
18	Acciones semánticas - stmtlist . . . . .	24
19	Acciones semánticas - stmt . . . . .	25
20	Acciones semánticas - caselist . . . . .	27
21	Acciones semánticas - block . . . . .	27
22	Acciones semánticas - controlSymbol . . . . .	28
23	Acciones semánticas - if . . . . .	28
24	Acciones semánticas - for . . . . .	29
25	Acciones semánticas - switch . . . . .	30
26	Acciones semánticas - repeat . . . . .	30
27	Acciones semánticas - while . . . . .	31
28	Acciones semánticas - cond . . . . .	31
29	Acciones semánticas - asgn . . . . .	31
30	Acciones semánticas - clear . . . . .	32
31	Acciones semánticas - plusplus . . . . .	32
32	Acciones semánticas - minusminus . . . . .	32
33	Acciones semánticas - place . . . . .	33
34	Acciones semánticas - print . . . . .	33
35	Acciones semánticas - read . . . . .	33
36	Acciones semánticas - writestring . . . . .	33
37	Acciones semánticas - readstring . . . . .	34
38	Acciones semánticas - exp (STRING) . . . . .	34
39	Acciones semánticas - exp (NUMBER) . . . . .	34

40	Acciones semánticas - listOfExp . . . . .	38
41	Acciones semánticas - restOfListOfExp . . . . .	38
42	Método <i>void lp::AssignmentStmt::evaluate()</i> . . . . .	42
43	Clase <i>CaseStmt : public Statement</i> . . . . .	43
44	Método <i>void lp::CaseStmt::evaluate()</i> . . . . .	44
45	Clase <i>ClearStmt : public Statement</i> . . . . .	45
46	Método <i>void lp::ClearStmt::print()</i> . . . . .	46
47	Método <i>void lp::ClearStmt::evaluate()</i> . . . . .	46
48	Clase <i>ForStmt : public Statement</i> . . . . .	46
49	Método <i>void lp::ForStmt::print()</i> . . . . .	47
50	Método <i>void lp::ForStmt::evaluate()</i> . . . . .	48
51	Clase <i>IfStmt : public Statement</i> . . . . .	49
52	Método <i>void lp::IfStmt::print()</i> . . . . .	51
53	Método <i>void lp::IfStmt::evaluate()</i> . . . . .	51
54	Clase <i>MinusEqualStmt : public Statement</i> . . . . .	52
55	Método <i>void lp::MinusEqualStmt::print()</i> . . . . .	53
56	Método <i>void lp::MinusEqualStmt::evaluate()</i> . . . . .	53
57	Clase <i>MinusMinusNode : public Statement</i> . . . . .	54
58	Método <i>void lp::MinusMinusNode::print()</i> . . . . .	55
59	Método <i>void lp::MinusMinusNode::evaluate()</i> . . . . .	55
60	Clase <i>PlaceStmt : public Statement</i> . . . . .	56
61	Método <i>void lp::PlaceStmt::print()</i> . . . . .	56
62	Método <i>void lp::PlaceStmt::evaluate()</i> . . . . .	57
63	Clase <i>PlusEqualStmt : public Statement</i> . . . . .	57
64	Método <i>void lp::PlusEqualStmt::print()</i> . . . . .	58
65	Método <i>void lp::PlusEqualStmt::evaluate()</i> . . . . .	58
66	Clase <i>PlusPlusNode : public Statement</i> . . . . .	59
67	Método <i>void lp::PlusPlusNode::print()</i> . . . . .	60
68	Método <i>void lp::PlusPlusNode::evaluate()</i> . . . . .	60
69	Clase <i>ReadStringStmt : public Statement</i> . . . . .	61
70	Método <i>void lp::ReadStringStmt::print()</i> . . . . .	62
71	Método <i>void lp::ReadStringStmt::evaluate()</i> . . . . .	62
72	Clase <i>RepeatStmt : public Statement</i> . . . . .	63
73	Método <i>void lp::RepeatStmt::print()</i> . . . . .	64
74	Método <i>void lp::RepeatStmt::evaluate()</i> . . . . .	64
75	Clase <i>SwitchStmt : public Statement</i> . . . . .	65
76	Método <i>void lp::SwitchStmt::evaluate()</i> . . . . .	66
77	Clase <i>WhileStmt : public Statement</i> . . . . .	68
78	Método <i>void lp::WhileStmt::print()</i> . . . . .	69
79	Método <i>void lp::WhileStmt::evaluate()</i> . . . . .	70
80	Clase <i>WriteStringStmt : public Statement</i> . . . . .	70
81	Método <i>void lp::WriteStringStmt::print()</i> . . . . .	71
82	Método <i>void lp::WriteStringStmt::evaluate()</i> . . . . .	71

83	Clase <i>DivisionEnteraNode</i> : <i>public NumericOperatorNode</i> . . . . .	74
84	Método <i>void lp::DivisionEnteraNode::print()</i> . . . . .	74
85	Método <i>void lp::DivisionEnteraNode::evaluate()</i> . . . . .	74
86	Clase <i>Concat</i> : <i>public StringOperatorNode</i> . . . . .	75
87	Método <i>void lp::Concat::print()</i> . . . . .	76
88	Método <i>void lp::Concat::evaluate()</i> . . . . .	76
89	Clase <i>StringNode</i> : <i>public ExpNode</i> . . . . .	77
90	Método <i>void lp::StringNode::getType()</i> . . . . .	78
91	Método <i>void lp::StringNode::print()</i> . . . . .	78
92	Método <i>void lp::StringNode::evaluate()</i> . . . . .	78
93	Clase <i>StringOperatorNode</i> : <i>public OperatorNode</i> . . . . .	78
94	Método <i>void lp::StringOperatorNode::getType()</i> . . . . .	79
95	Control del modo de ejecución. . . . .	80
96	Ejecución del intérprete a partir de un fichero. . . . .	85
97	Ejemplos - <i>menu.e</i> . . . . .	86
98	Ejemplos - <i>conversion.e</i> . . . . .	89
99	Ejemplos - <i>test_#_y_mayusculas.e</i> . . . . .	90
100	Ejemplos - <i>test_booleano.e</i> . . . . .	91
101	Ejemplos - <i>test_cadenas.e</i> . . . . .	92
102	Ejemplos - <i>test_casos.e</i> . . . . .	93
103	Ejemplos - <i>test_error.txt</i> . . . . .	94
104	Ejemplos - <i>test_numeros.e</i> . . . . .	94
105	Ejemplos - <i>test_operadores_aritmeticos.e</i> . . . . .	95
106	Ejemplos - <i>test_operadores_relacionales.e</i> . . . . .	96
107	Ejemplos - <i>test_palabras_reservadas.e</i> . . . . .	97
108	Ejemplos - <i>test_para.e</i> . . . . .	100
109	Ejemplos - <i>test_repetir.e</i> . . . . .	100
110	Ejemplos - <i>test_si.e</i> . . . . .	101

## Índice de cuadros

1	Zona de declaraciones - Expresiones regulares. . . . .	12
2	Zona de declaraciones - Control de errores. . . . .	13
3	Símbolos terminales precedidos por <i>%token</i> . . . . .	18
4	Símbolos terminales precedidos por <i>%right</i> . . . . .	18
5	Símbolos terminales precedidos por <i>%left</i> . . . . .	19
6	Símbolos terminales precedidos por <i>%nonassoc</i> . . . . .	19
7	Símbolos no terminales de tipo <i>&lt;expNode&gt;</i> . . . . .	19
8	Símbolos no terminales de tipo <i>&lt;parameters&gt;</i> . . . . .	19
9	Símbolos no terminales de tipo <i>&lt;stmts&gt;</i> . . . . .	19
10	Símbolos no terminales de tipo <i>&lt;st&gt;</i> . . . . .	20
11	Símbolos no terminales de tipo <i>&lt;prog&gt;</i> . . . . .	20



# 1 Introducción

## 1.1 Descripción del trabajo

El trabajo de final de prácticas ha consistido en la construcción de un intérprete de pseudocódigo en el lenguaje español, haciendo uso de *Flex* y *Bison* y aplicando las técnicas aprendidas durante el desarrollo de las prácticas de la asignatura Procesadores de lenguajes, tanto en el ámbito léxico, como el semántico y el sintáctico.

## 1.2 Partes del documento

El documento se encuentra dividido en las siguientes partes:

- **Lenguaje de pseudocódigo:** donde encontraremos tanto los componentes léxicos como las sentencias usadas.
- **Tabla de símbolos:** muestra la información relacionada con el programa, entre las que destacan las variables, funciones, constantes, palabras clave; entre otras.
- **Análisis:**
  - **Análisis léxico:** obtención de *tokens* para el programa.
  - **Análisis sintáctico:** se reciben los componentes léxicos y se comprueban que cumplan las reglas sintácticas de lenguaje de programación.
- **AST:** se detallan tanto clases preestablecidas como las que han sido creadas y/o modificadas.
- **Funciones auxiliares:** cualquier tipo de función extra que haya sido necesitada para llevar acabo el correcto desarrollo del intérprete.
- **Modo de obtención del intérprete:** contiene una estructuración de los diversos directorios en los que se divide el intérprete, así como los ficheros usados en cada uno.
- **Modos de ejecución:** se explican las distintas formas de ejecución del intérprete.
- **Ejemplos:** sección dedicada a la puesta en práctica del intérprete, con diversos ejercicios ejemplificativos que muestran sus distintas funcionalidades.
- **Conclusiones:** evaluación del trabajo realizado, tomando en cosideración tanto puntos fuertes como puntos débiles.

## 2 Lenguaje de pseudocódigo

### 2.1 Componentes léxicos

Los componentes léxicos, también denominados *tokens*, son los elementos más simples con significado propio de un lenguaje de programación. Para su análisis, se ha hecho uso de una potente herramienta llamada *Flex*. Para ello, se ha desarrollado el fichero “*interpreter.l*”, en el que se detallan los siguientes componentes léxicos:

- **Palabras reservadas:**

- **Utilizadas en sentencias:**

- \* *leer, leer\_cadena*
    - \* *escribir, escribir\_cadena*
    - \* *si, entonces, si\_no, fin\_si*
    - \* *mientras, hacer, fin\_mientras*
    - \* *repetir, hasta, para, fin\_para, desde, paso*
    - \* *casos, valor, defecto, fin\_casos*
    - \* *verdadero, falso*

- **Otras:** van precedidas de #

- \* *#mod, #div*
    - \* *#o, #y, #no*
    - \* *#borrar, #lugar*

- **Identificadores:**

- **Características:**

- \* Compuestos por una serie de letras, dígitos y el subrayado.
    - \* Deben comenzar por una letra.
    - \* No podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.

- **Identificadores válidos:**

- \* *dato, dato\_1, dato\_1\_a*

- **Identificadores no válidos:**

- \* *\_dato, dato\_, dato\_1\_*

- **No se distinguirá entre mayúsculas ni minúsculas.**

- **Números:**

- Se utilizan números enteros, reales de punto fijo y reales con notación científica.
  - Todos ellos serán tratados conjuntamente como números.

- **Cadenas:**

- Estarán compuestas por una serie de caracteres delimitados por comillas simples:
  - \* *‘Ejemplo cadena’*
  - \* *‘Ejemplo de cadena con salto de línea \n y tabulador \t’*
- Deberán permitir la inclusión de la comilla simple utilizando la barra (\):
  - \* *‘Ejemplo de cadena con \'comillas\' simples’*
- **Operador de asignación:**
  - **Asignación:** :=
- **Operadores aritméticos:**
  - **Suma:** +
  - **Resta:** -
  - **Producto:** \*
  - **División:** /
  - **División entera:** #div
  - **Módulo:** #mod
  - **Potencia:** \*\*
- **Operadores alfanuméricos:**
  - **Concatenación:** ||
- **Operadores relacionales de números y cadenas:**
  - **Menor que:** <
  - **Menor o igual que:** ≤
  - **Mayor que:** >
  - **Mayor o igual que:** ≥
  - **Igual que:** =
  - **Distinto que:** <>
- **Operadores lógicos:**
  - **Disyunción lógica:** #o
  - **Conjunción lógica:** #y
  - **Negación lógica:** #no
- **Comentarios:**
  - **Varias líneas:** delimitados los símbolos << y >>
  - **Una línea:** Todo lo que siga al carácter @ hasta el final de la línea.
- **Punto y coma ";":** Se utiliza para indicar el fin de una sentencia.

## 2.2 Sentencias

Una sentencia es el conjunto de componentes léxicos que se interpretan conjuntamente para realizar una tarea.

- **Asignación:**

- **identificador := expresión numérica**

- \* Declara a identificador como una variable numérica y le asigna el valor de la expresión numérica.
    - \* Las expresiones numéricas se formarán con números, variables numéricas y operadores numéricos.

- **identificador := expresión alfanumérica**

- \* Declara a identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.
    - \* Las expresiones alfanuméricas se formarán con cadenas, variables alfanuméricas y el operador alfanumérico de concatenación (||).

- **Lectura:**

- **leer(identificador)**

- \* Declara a identificador como variable numérica y le asigna el número leído.

- **leer\_cadena(identificador)**

- \* Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).

- **Escritura:**

- **escribir(identificador)**

- \* El valor de la expresión numérica es escrito en la pantalla.

- **escribir\_cadena(expresión alfanumérica)**

- \* La cadena (sin comillas exteriores) es escrita en la pantalla.
    - \* Se debe permitir la interpretación de comandos de saltos de línea (\n) y tabuladores (\t) que puedan aparecer en la expresión alfanumérica.

- **Sentencias de control:**

- **Sentencia condicional simple**

*si condición*

*entonces lista de sentencias*

*fin\_si;*

- **Sentencia condicional compuesta**
  - si condición*
  - entonces lista de sentencias*
  - si\_no lista de sentencias*
  - fin\_si;*
- **Bucle “mientras”**
  - mientras condición hacer*
  - lista de sentencias*
  - fin\_mientras;*
- **Bucle “repetir”**
  - repetir*
  - lista de sentencias*
  - hasta condición;*
- **Bucle “para”**
  - para identificador*
  - desde expresión numérica 1*
  - hasta expresión numérica 2*
  - [paso expresión numérica 3]*
  - hacer*
  - lista de sentencias*
  - fin\_para;*
- **Sentencia “casos”**
  - casos (expresión)*
  - valor v1: ...*
  - valor v2: ...*
  - ...*
  - defecto: ....*
  - fin\_casos;*
- **Comandos especiales:**
  - **#borrar**
    - \* Borra la pantalla.
  - **#lugar(expresión numérica1, expresión numérica2)**
    - \* Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.
- **Ampliaciones:**
  - **Operadores unarios:** ++, --
  - **Operadores aritméticos y de asignación:** +=, -=
- **Observaciones:**

- Se permite que una variable pueda cambiar de tipo durante el tiempo de ejecución del intérprete.

### 3 Tabla de símbolos

#### 3.1 Descripción

Una tabla de símbolos es una estructura de datos en la que cada símbolo del código fuente de un programa se ve asociado con cierta información, entre la que se puede encontrar el tipo de dato, ubicación, procedimientos; entre otros.

En nuestro caso, la jerarquía que presenta dicha tabla de símbolos es la siguiente:

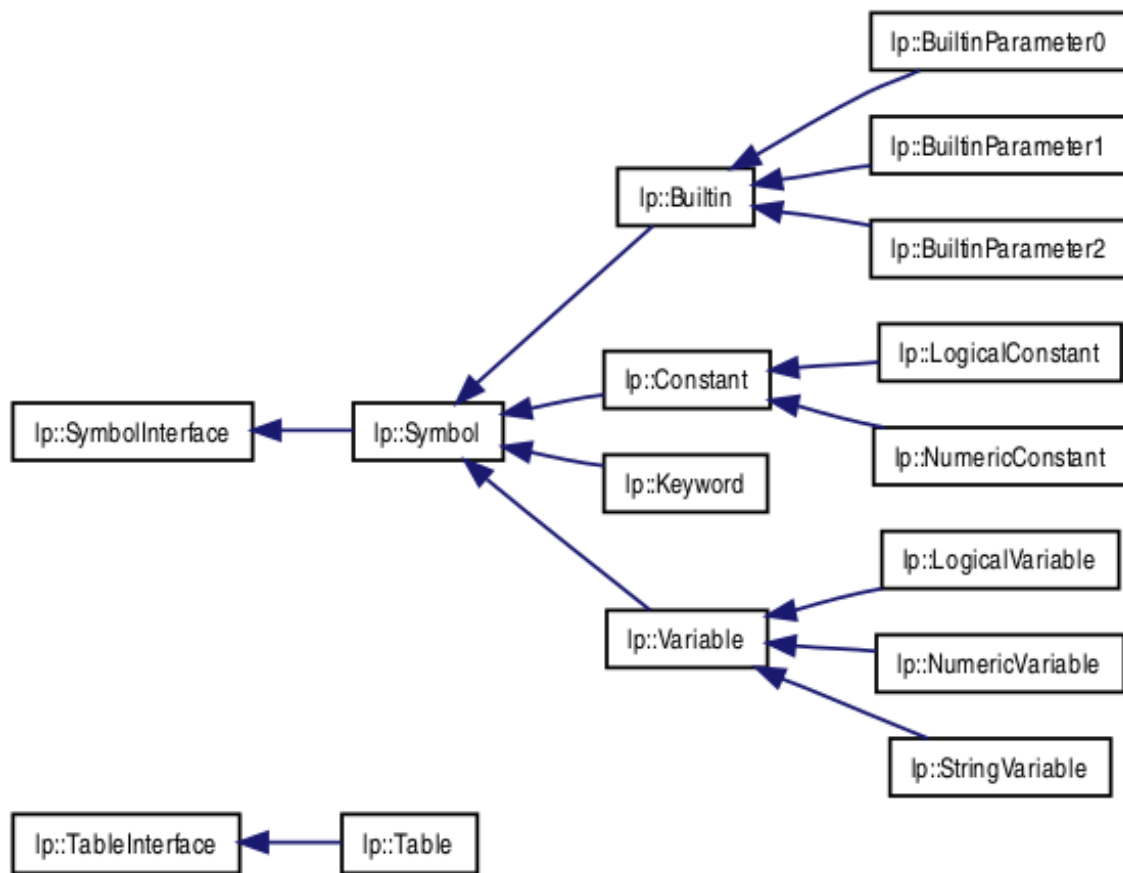


Figura 1: Jerarquía de clases de SymbolInterface y TableInterface.

Empezando desde arriba encontramos las clases *BuilingParameter*, que permiten el uso de 0, 1 ó 2 parámetros en las funciones; *Logical* y *Numeric*, que en el caso de ser constantes permiten crear variables que no pueden ser modificadas. De todas ellas, *StringVariable* es la más relevante en este trabajo pues es la única desarrollada por los autores del informe. Como su nombre indica, permite crear variables de tipo **string** (o **cadena** en español). Al ser una variable como las lógicas o las numéricas, hereda de *Variable*.

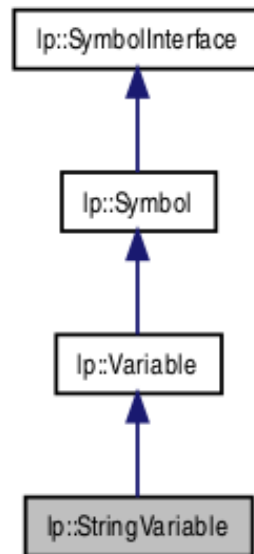


Figura 2: Jerarquía de herencia de la clase StringVariable.

El código esqueleto de la clase ha sido el de *numericVariable.hpp* y *numericVariable.cpp*, los cuales se han modificado para acomodar el uso de cadenas en el intérprete en español.



## 4 Análisis léxico

### 4.1 Descripción

Tal y como se ha comentado con anterioridad, el análisis léxico es la única fase que tiene contacto con el código del programa, de tal manera que favorece la modularidad y la interactividad. De esta manera, lee el programa carácter por carácter obteniendo los distintos *tokens*.

Dividiremos esta sección en dos subsecciones más, de tal manera que quede bien diferenciadas entre sí. Dichas subsecciones son el **área de declaraciones** y el **área de reglas**.

#### 4.1.1 Área de declaraciones

Se encuentran las distintas expresiones regulares utilizadas para la obtención de los distintos componentes léxicos o *tokens*. Además, también permiten hacer una buena gestión y controlar los distintos errores que se puedan ocasionar:

Nombre	Expresión regular
DIGIT	[0-9]
LETTER	[a-zA-Z]
NUMBER1	{DIGIT}+(\.{DIGIT})?
NUMBER2	{DIGIT}(\.{DIGIT})?(E[+-]?{DIGIT})?
IDENTIFIER	{LETTER}({LETTER}+ {DIGIT}+ (-{LETTER})+ (-{DIGIT})+)*
INLINE_COMMENT	@(.)*\$

Cuadro 1: Zona de declaraciones - Expresiones regulares.

- DIGIT: detecta un solo número del 0 al 9.
- LETTER: detecta una sola letra, sea minúscula o mayúscula.
- NUMBER1: formado por varios DIGIT, detecta números de varias cifras así como decimales.
- NUMBER2: es una extensión de NUMBER1 que permite detectar números en notación científica.
- IDENTIFIER: formado por LETTER y DIGIT, es capaz de detectar palabras que empiecen por una letra y tengan uno o varios números y guiones bajos.
- INLINE\_COMMENT: permite la detección de comentarios de una sola línea.

Nombre	Expresión regular
INVALID_NUMBER INVALID_IDENTIFIER	{DIGIT}+(\.)(E[+]?{DIGIT}+)? {DIGIT}+(\.){DIGIT}+(E[+]?){DIGIT}+ {DIGIT}+(\.)(E[+]?) ({LETTER}\- {DIGIT})({LETTER}+ {DIGIT}+(\.)+)+({LETTER} {DIGIT} \-)?

Cuadro 2: Zona de declaraciones - Control de errores.

- INVALID\_NUMBER: recoge ciertas formas en las que un número puede estar mal escrito, la mayoría de ellas relacionadas con un uso incorrecto de la notación científica.
- INVALID\_IDENTIFIER: de la misma forma que con los números, tenemos una expresión regular para los identificadores mal escritos. Este detecta palabras como por ejemplo \_dato, dato\_ o dato\_\_1.

#### 4.1.2 Área de reglas

En esta sección podremos observar tanto cómo se ligan las expresiones regulares como la simbología asociada a operadores y otros tipos de caracteres, cada uno de ellos asociado a un *token*. Para ello, veremos cómo se produce este reconocimiento<sup>1</sup>:

Los tabuladores y espacios en blanco se pasan por alto en el intérprete.

Listing 1: Área de reglas - Espacios en blanco y tabuladores.

```
[ \t] { ; }
```

Los saltos de línea incrementan el contador de líneas.

Listing 2: Área de reglas - Salto de línea.

```
\n { lineNumber++; }
```

Con la primera comilla detectada se inicia el estado cadena.

Listing 3: Área de reglas - Reconocimiento de inicio de cadena.

```
" " { BEGIN STRING_ST; }
```

Al reconocer una segunda comilla (que no esté precedida de un backslash) se finaliza la cadena, se omite el carácter de fin de cadena, se vuelve al estado por defecto y se termina por devolver dicha cadena al fichero de *Bison*.

<sup>1</sup>Las siguientes reglas vienen ordenadas según el orden de codificación.

Listing 4: Área de reglas - Reconocimiento de final de cadena.

```

<STRING_ST>"'"
{
    BEGIN 0;
    yytext[yyldeng-1] = '\0';
    yylval.string = yytext;
    return STRING;
}

```

De la misma forma que con las cadenas, al reconocer el doble signo <<, se inicia el estado de un comentario de varias líneas.

Listing 5: Área de reglas - Reconocimiento de inicio de comentario de varias líneas.

```

"<<"
{
    yymore();
    BEGIN(Q1);
}

```

Si se vuelve a encontrar un doble signo <<, se añade al comentario.

Listing 6: Área de reglas - Estado Q1 - Reconocimiento de inicio de comentario de varias líneas.

```

<Q1>[^<<] { yymore(); }

```

Es al encontrar un doble signo >> cuando realmente se termina el estado comentario y se vuelve al inicial.

Listing 7: Área de reglas - Estado Q1 - Reconocimiento de final de comentario de varias líneas.

```

<Q1>">>" { BEGIN 0; }

```

Al encontrar el símbolo del punto y coma, se devuelve el token correspondiente.

Listing 8: Área de reglas - Punto y coma.

```

";" { return SEMICOLON; }

```

Al encontrar el símbolo de la coma, se devuelve el token correspondiente.

Listing 9: Área de reglas - Coma.

```

",," { return COMMA; }

```

Cuando se detecta un número decimal o en notación científica, se transforma a flotante y se envía a *Bison*.

Listing 10: Área de reglas - Valores numéricos.

```
{NUMBER1}|{NUMBER2}
{
    yyval.number = atof(yytext);
    return NUMBER;
}
```

Esta regla, desarrollada por el profesor de la asignatura y actualizada por los autores de la práctica, maneja la detección de identificadores. Lo primero que se hace es pasar a minúscula todas las letras de la palabra, para luego buscarla en la tabla de símbolos. Si no se encuentra en ella, se inserta un puntero a la variable numérica que la representa. Si por el contrario el identificador ya está presente en la tabla, se devuelve su token.

Listing 11: Área de reglas - Identificadores.

```
{IDENTIFIER}
{
    std::string identifier(yytext);
    for (size_t i = 0; i < identifier.size(); i++)
    {
        identifier[i] = tolower(identifier[i]);
    }

    yyval.identifier = strdup(identifier.c_str());

    if (table.lookupSymbol(identifier) == false)
    {
        lp::NumericVariable *n = new lp::NumericVariable(
            identifier, VARIABLE, UNDEFINED, 0.0);

        table.installSymbol(n);

        return VARIABLE;
    }
    else
    {
        lp::Symbol *s = table.getSymbol(identifier);

        return s->getToken();
    }
}
```

Al detectar una comentario en línea no se hace nada.

Listing 12: Área de reglas - Comentarios de una línea.

```
{INLINE_COMMENT} { ; }
```

Todas las reglas siguientes permiten identificar los operadores aritméticos, los de asignación y los relacionales. Además de los corchetes, paréntesis y ciertas palabras reservadas que empiezan

por # y son identificadas sin importar que estén en mayúsculas o minúsculas. Se han agrupado todas ellas ya que lo único que hacen es devolver el token correspondiente de cada una de sus expresiones regulares.

Listing 13: Área de reglas - Operadores y otros.

```
"-" { return MINUS; }
"+" { return PLUS; }
"++" { return PLUSPLUS; }
"--" { return MINUSMINUS; }
"*" { return MULTIPLICATION; }
"/" { return DIVISION; }
(?i:#div) { return ENTIRE_DIVISION; }
 "(" { return LPAREN; }
 ")" { return RPAREN; }
(?i:#mod) { return MODULO; }
(?i:#borrar) { return CLEAR; }
(?i:#lugar) { return PLACE; }
" **" { return POWER; }
"+:=" { return PLUS_EQUAL; }
"-:=" { return MINUS_EQUAL; }
":=" { return ASSIGNMENT; }
"=" { return EQUAL; }
"<>" { return NOT_EQUAL; }
">=" { return GREATER_OR_EQUAL; }
"<=" { return LESS_OR_EQUAL; }
">" { return GREATER_THAN; }
"<" { return LESS_THAN; }
(?i:#no) { return NOT; }
(?i:#o) { return OR; }
(?i:#y) { return AND; }
"{" { return LETFCURLYBRACKET; }
"}" { return RIGHTCURLYBRACKET; }
"|" { return CONCAT; }
":" { return COLON; }
```

También se destacan otras reglas para el control de errores que solo devuelven un mensaje de error:

Listing 14: Área de reglas de control de errores - Valores numéricos no válidos.

```
{INVALID_NUMBER} {warning("Lexical error: Not a valid number
.", yytext); }
```

Listing 15: Área de reglas de control de errores - Identificadores no válidos.

```
{INVALID_IDENTIFIER} { warning("Lexical error: Not a valid
identifier.", yytext); }
```

Listing 16: Área de reglas de control de errores - Otros.

```
.
{
    BEGIN(ERROR);
    yymore();
}

<ERROR>[^0-9+\\-*/()\\^% \\t\\n\\;a-zA-Z=<>!&] { yymore(); }

<ERROR>(\\.|\\n)
{
    yyless(yytext-1);
    warning("Lexical error", yytext);
    BEGIN(INITIAL);
}
```

## 5 Análisis sintáctico

### 5.1 Símbolos terminales

Se corresponden con los componentes léxicos de la gramática. Se clasificarán según su directiva<sup>2</sup>:

%token	
SEMICOLON	PRINT
READ	IF
ELSE	WHILE
CLEAR	PLACE
THEN	ENDIF
DO	ENDWHILE
REPEAT	UNTIL
FOR	STEP
TO	ENDFOR
FROM	READSTRING
WRITESTRING	SWITCH
CASE	DEFAULT
ENDSWITCH	COLON
PLUSPLUS	MINUSMINUS
LEFTCURLYBRACKET	RIGHTCURLYBRACKET
COMMA	<number>
<string>	<logic>
<identifier>	

Cuadro 3: Símbolos terminales precedidos por *%token*.

%right	
ASSIGNMENT	PLUS_EQUAL
MINUS_EQUAL	POWER

Cuadro 4: Símbolos terminales precedidos por *%right*.

<sup>2</sup>Puede visualizar todos los elementos de esta sección en el fichero fuente *interpreter.y*

%left	
OR	AND
NOT	PLUS
MINUS	CONCAT
MULTIPLICATION	DIVISION
MODULO	ENTIRE_DIVISION
LPAREN	RPARENT

Cuadro 5: Símbolos terminales precedidos por %left.

%nonassoc	
UNARY	PLSUPLUS
MINUSMINUS	GREATER_OR_EQUAL
LESS_OR_EQUAL	GREATER_THAN
LESS_THAN	EQUAL
NOT_EQUAL	

Cuadro 6: Símbolos terminales precedidos por %nonassoc.

## 5.2 Símbolos no terminales

De igual manera se clasificarán, en este caso, por su tipo de dato:

<expNode>	
exp	cond

Cuadro 7: Símbolos no terminales de tipo &lt;expNode&gt;.

<parameters>	
listOfExp	restOfListOfExp

Cuadro 8: Símbolos no terminales de tipo &lt;parameters&gt;.

<stmts>	
stmtlist	

Cuadro 9: Símbolos no terminales de tipo &lt;stmts&gt;.



<st>	
stmt	asgn
print	read
if	while
block	repeat
for	clear
place	writestring
readstring	switch
plusplus	minusminus

Cuadro 10: Símbolos no terminales de tipo *<st>*.

<progr>
program

Cuadro 11: Símbolos no terminales de tipo *<prog>*.

### 5.3 Reglas de producción

Las reglas de producción definidas en nuestra gramática son las siguientes:

1.  $\text{program} \rightarrow \text{stmtlist}$
2.  $\text{stmtlist} \rightarrow \epsilon$
3.  $\text{stmtlist} \rightarrow \text{stmtlist stmt}$
4.  $\text{stmtlist} \rightarrow \text{stmtlist error}$
5.  $\text{stmt} \rightarrow \text{SEMICOLON}$
6.  $\text{stmt} \rightarrow \text{asgn SEMICOLON}$
7.  $\text{stmt} \rightarrow \text{print SEMICOLON}$
8.  $\text{stmt} \rightarrow \text{read SEMICOLON}$
9.  $\text{stmt} \rightarrow \text{if}$
10.  $\text{stmt} \rightarrow \text{while}$
11.  $\text{stmt} \rightarrow \text{repeat}$
12.  $\text{stmt} \rightarrow \text{for}$

13. `stmt`  $\rightarrow$  `switch`
14. `stmt`  $\rightarrow$  `block`
15. `stmt`  $\rightarrow$  `clear SEMICOLON`
16. `stmt`  $\rightarrow$  `place SEMICOLON`
17. `stmt`  $\rightarrow$  `plusplus SEMICOLON`
18. `stmt`  $\rightarrow$  `minusminus SEMICOLON`
19. `stmt`  $\rightarrow$  `writestring SEMICOLON`
20. `stmt`  $\rightarrow$  `readstring SEMICOLON`
21. `stmt`  $\rightarrow$  `PLUS_EQUAL SEMICOLON`
22. `stmt`  $\rightarrow$  `MINUS_EQUAL SEMICOLON`
23. `caselist`  $\rightarrow \epsilon$
24. `caselist`  $\rightarrow$  `caselist CASE exp COLON stmtlist`
25. `block`  $\rightarrow$  `LEFTCURLYBRACKET stmtlist RIGHTCURLYBRACKET`
26. `controlSymbol`  $\rightarrow \epsilon$
27. `if`  $\rightarrow$  `IF controlSymbol cond THEN stmtlist ENDIF SEMICOLON`
28. `if`  $\rightarrow$  `IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF SEMICOLON`
29. `for`  $\rightarrow$  `FOR controlSymbol VARIABLE FROM exp UNTIL exp DO stmtlist ENDFOR SEMICOLON`
30. `for`  $\rightarrow$  `FOR controlSymbol VARIABLE FROM exp UNTIL exp STEP exp DO stmtlist ENDFOR SEMICOLON`
31. `switch`  $\rightarrow$  `SWITCH controlSymbol cond caselist ENDSWITCH SEMICOLON`
32. `switch`  $\rightarrow$  `SWITCH controlSymbol cond caselist DEFAULT COLON stmtlist ENDSWITCH SEMICOLON`
33. `repeat`  $\rightarrow$  `REPEAT controlSymbol stmtlist UNTIL cond SEMICOLON`
34. `while`  $\rightarrow$  `WHILE controlSymbol cond DO stmtlist ENDWHILE SEMICOLON`
35. `cond`  $\rightarrow$  `LPAREN exp RPAREN`
36. `asgn`  $\rightarrow$  `VARIABLE ASSIGNMENT exp`
37. `asgn`  $\rightarrow$  `VARIABLE PLUS_EQUAL exp`

- 38. `asgn`  $\rightarrow$  `VARIABLE MINUS_EQUAL exp`
- 39. `clear`  $\rightarrow$  `CLEAR`
- 40. `plusplus`  $\rightarrow$  `PLUSPLUS VARIABLE`
- 41. `minuminus`  $\rightarrow$  `MINUSMINUS VARIABLE`
- 42. `place`  $\rightarrow$  `PLACE LPAREN exp COMMA exp RPAREN`
- 43. `print`  $\rightarrow$  `PRINT exp`
- 44. `read`  $\rightarrow$  `READ LPAREN VARIABLE RPAREN`
- 45. `writestring`  $\rightarrow$  `WRITESTRING LPAREN exp RPAREN`
- 46. `readstring`  $\rightarrow$  `READSTRING LPAREN VARIABLE RPAREN`
- 47. `exp`  $\rightarrow$  `STRING`
- 48. `exp`  $\rightarrow$  `exp CONCAT exp`
- 49. `exp`  $\rightarrow$  `NUMBER`
- 50. `exp`  $\rightarrow$  `exp PLUS exp`
- 51. `exp`  $\rightarrow$  `exp MINUS exp`
- 52. `exp`  $\rightarrow$  `exp MULTIPLICATION exp`
- 53. `exp`  $\rightarrow$  `exp DIVISION exp`
- 54. `exp`  $\rightarrow$  `exp ENTIRE_DIVISION exp`
- 55. `exp`  $\rightarrow$  `LPAREN exp RPAREN`
- 56. `exp`  $\rightarrow$  `PLUS exp %prec UNARY`
- 57. `exp`  $\rightarrow$  `MINUS exp %prec UNARY`
- 58. `exp`  $\rightarrow$  `exp MODULO exp`
- 59. `exp`  $\rightarrow$  `exp POWER exp`
- 60. `exp`  $\rightarrow$  `VARIABLE`
- 61. `exp`  $\rightarrow$  `CONSTANT`
- 62. `exp`  $\rightarrow$  `BULTING LPAREN listOfExp RPAREN`
- 63. `exp`  $\rightarrow$  `exp GREATER_THAN exp`

64.  $\text{exp} \rightarrow \text{exp GREATER\_OR\_EQUAL exp}$
65.  $\text{exp} \rightarrow \text{exp LESS\_THAN exp}$
66.  $\text{exp} \rightarrow \text{exp LESS\_OR\_EQUAL exp}$
67.  $\text{exp} \rightarrow \text{exp EQUAL exp}$
68.  $\text{exp} \rightarrow \text{exp NOT\_EQUAL exp}$
69.  $\text{exp} \rightarrow \text{exp AND exp}$
70.  $\text{exp} \rightarrow \text{exp OR exp}$
71.  $\text{listOfExp} \rightarrow \epsilon$
72.  $\text{listOfExp} \rightarrow \text{exp restOfListOfExp}$
73.  $\text{restOfListOfExp} \rightarrow \epsilon$
74.  $\text{restOfListOfExp} \rightarrow \text{COMMA exp restOfListOfExp}$

### 5.3.1 Control de errores

Las reglas de producción enfocadas al control de errores son las siguientes:

1.  $\text{if} \rightarrow \text{IF controlSymbol THEN stmtlist ENDIF SEMICOLON}$
2.  $\text{if} \rightarrow \text{IF controlSymbol THEN stmtlist ELSE stmtlist ENDIF SEMICOLON}$
3.  $\text{switch} \rightarrow \text{SWITCH controlSymbol caselist ENDSWITCH SEMICOLON}$
4.  $\text{switch} \rightarrow \text{SWITCH controlSymbol caselist DEFAULT COLON stmtlist ENDSWITCH SEMICOLON}$
5.  $\text{while} \rightarrow \text{WHILE controlSymbol DO stmtlist ENDWHILE SEMICOLON}$
6.  $\text{asgn} \rightarrow \text{CONSTANT ASSIGNMENT exp}$
7.  $\text{asgn} \rightarrow \text{CONSTANT ASSIGNMENT asgn}$
8.  $\text{read} \rightarrow \text{READ LPAREN CONSTANT RPAREN}$
9.  $\text{readstring} \rightarrow \text{READSTRING LPAREN CONSTANT RPAREN}$

## 5.4 Acciones semánticas

Se han definido un total de 25 acciones semánticas:

La acción semántica de *program* crea la clase que manejará el árbol AST y lo asigna a la raíz.

Listing 17: Acciones semánticas - *program*

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

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

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

*stmtlist* puede utilizar hasta tres acciones semánticas:

- Regla épsilon: crea una lista vacía de sentencias.
- Nueva stmt: añade una nueva sentencia a la lista y, en función de si está activado el modo interactivo o no, evalúa la lista y borra el código AST.
- Error: solo copia la lista de sentencias si ocurre un error.

Listing 18: Acciones semánticas - *stmtlist*

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

```

De similar manera que con *stmtlist*, *stmt* también maneja un gran número de acciones semánticas, aunque ninguna se usa en nuestra práctica.

#### Listing 19: Acciones semánticas - stmt

```

stmt: SEMICOLON /* Empty statement: ";" */
{
  // Create a new empty statement node
  $$ = new lp::EmptyStmt();
}

| asgn SEMICOLON
{
  // Default action
  // $$ = $1;
}

| print SEMICOLON
{
  // Default action
  // $$ = $1;
}

| read SEMICOLON
{
  // Default action
  // $$ = $1;
}

| if
{
  // Default action
  // $$ = $1;
}

| while
{
  // Default action

```

```
        // $$ = $1;
    }

    | repeat
    {
        // Default action
        // $$ = $1;
    }

    | for
    {
        // Default action
        // $$ = $1;
    }

    | switch
    {
        // Default action
        // $$ = $1;
    }

    | block
    {
        // Default action
        // $$ = $1;
    }

    | clear SEMICOLON
    {
        // Default action
        // $$ = $1;
    }

    | place SEMICOLON
    {
        // Default action
        // $$ = $1;
    }

    | plusplus SEMICOLON
    {
        // Default action
        // $$ = $1;
    }

    | minusminus SEMICOLON
    {
        // Default action
        // $$ = $1;
    }

    | writestring SEMICOLON
    {
```

```

        // Default action
        // $$ = $1;
    }

    | readstring SEMICOLON
    {
        // Default action
        // $$ = $1;
    }

    | PLUS_EQUAL SEMICOLON
    {
        // Create a new "plus equal" node
        // $$ = $1;
    }

    | MINUS_EQUAL SEMICOLON
    {
        // Create a new "minus equal" node
        // $$ = $1;
    };

```

Imitando a los *stmtlist* hemos decidido crear *caselist*, que maneja los casos en el bucle *switch*. Está formado por dos acciones semánticas:

- Regla épsilon: crea una lista vacía de casos.
- Nuevo caso: al detectar un nuevo caso, lo añade a la lista.

#### Listing 20: Acciones semánticas - caselist

```

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

|      caselist CASE exp COLON stmtlist
{
    // Get the list of expressions
    $$ = $1;

    // Insert the expression in the list of expressions
    $$->push_back(new lp::CaseStmt($3, $5));
};

```

Crea un nuevo bloque de nodos de sentencias.

#### Listing 21: Acciones semánticas - block

```

block: LETFCURLYBRACKET stmtlist RIGHTCURLYBRACKET

```



```

{
    // Create a new block of statements node
    $$ = new lp::BlockStmt($2);
};

```

*controlsymbol* incrementa la variable de control, que permite usar correctamente ciertos bucles y funciones condicionales como *para* y *si*.

#### Listing 22: Acciones semánticas - controlSymbol

```

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

```

Estas cuatro acciones semánticas permiten el inicio de la interpretación del condicional *si-entonces*.

- Condicional 1: *si* sin consecuente. Primero crea el nodo y luego disminuye en uno la variable de control.
- Condicional 2: *si* formado por un antecedente y un consecuente.
- Error 1: se acciona si se detecta un *si* 1 sin condicional.
- Error 2: se acciona si se detecta un *si* 2 sin condicional.

#### Listing 23: Acciones semánticas - if

```

if: /* Simple conditional statement */
IF controlSymbol cond THEN stmtlist ENDIF SEMICOLON
{
    // 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 ENDIF
  SEMICOLON
{
    // Create a new if statement node
    $$ = new lp::IfStmt($3, $5, $7);

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

```

```

| IF controlSymbol THEN stmtlist ENDIF SEMICOLON
{
    execerror("Semantic error in  \"if statement\": there is no
        condition \",\"");
}

| IF controlSymbol THEN stmtlist ELSE stmtlist ENDIF SEMICOLON
{
    execerror("Semantic error in  \"if statement\": there is no
        condition \",\"");
};

```

De forma similar al condicional *si*, el bucle *para* se detecta, crea un nodo con la información necesaria y disminuye en uno la variable de control.

- Bucle 1: para la sentencia *para* por defecto.
- Bucle 2: para la sentencia *para* que incluye a la expresión de *paso*.

#### Listing 24: Acciones semánticas - for

```

for:
FOR controlSymbol VARIABLE FROM exp UNTIL exp DO stmtlist ENDFOR
SEMICOLON
{
    // Create a new for statement node
    $$ = new lp::ForStmt($3, $5, $7, $9);

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

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

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

```

Las acciones semánticas del bucle *casos* tienen una estructura similar a la del resto de bucles, donde se detecta una estructura de identificadores, expresiones y condicionales, estos se mandan a un nuevo nodo *SwitchStmt* del árbol AST. Por último se disminuye en uno la variable de control.

- Bucle 1: para la sentencia *casos* por defecto.

- Bucle 2: para la sentencia *casos* incluyendo el paso por defecto.
- Error 1: se acciona si se detecta un *casos* 1 sin condicional.
- Error 2: se acciona si se detecta un *casos* 2 sin condicional.

#### Listing 25: Acciones semánticas - switch

```
switch:
SWITCH controlSymbol cond caselist ENDSWITCH SEMICOLON
{
    // Create a new switch statement node
    $$ = new lp::SwitchStmt($3, $4);

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

| SWITCH controlSymbol cond caselist DEFAULT COLON stmtlist
  ENDSWITCH SEMICOLON
{
    // Create a new switch statement node
    $$ = new lp::SwitchStmt($3, $4, $7);

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

| SWITCH controlSymbol caselist ENDSWITCH SEMICOLON
{
    execerror("Semantic error in \"switch statement\": there is
              no condition \",\"");
}

| SWITCH controlSymbol caselist DEFAULT COLON stmtlist ENDSWITCH
  SEMICOLON
{
    execerror("Semantic error in \"switch statement\": there is
              no condition \",\"");
};
```

Acción similar a los otros bucles pero para *repetir*.

#### Listing 26: Acciones semánticas - repeat

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

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

Acción similar a los otros bucles pero para *mientras*.

- Sentencia *mientras*.
- Error que se acciona si se detecta un *mientras* sin condicional.

#### Listing 27: Acciones semánticas - while

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

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

| WHILE controlSymbol DO stmtlist ENDWHILE SEMICOLON
{
    execerror("Semantic error in \"while statement\": there is
              no condition \",\"");
};
```

Esta acción semántica de la regla cond únicamente reconoce expresiones que van entre paréntesis, asignandolas al resultado final \$\$.

#### Listing 28: Acciones semánticas - cond

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

Aquí se agrupan las reglas relativas al símbolo no terminal *asgn*, las cuales asignan valores a una variable en función del operador correspondiente.

#### Listing 29: Acciones semánticas - asgn

```
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);
}

| VARIABLE PLUS_EQUAL exp
{
```

```

    // Create a new assignment node
    $$ = new lp::PlusEqualStmt($1, $3);
}

| VARIABLE MINUS_EQUAL exp
{
    // Create a new assignment node
    $$ = new lp::MinusEqualStmt($1, $3);
}

| CONSTANT ASSIGNMENT exp
{
    execerror("Semantic error in assignment: it is not allowed
              to modify a constant ", $1);
}

| CONSTANT ASSIGNMENT asgn
{
    execerror("Semantic error in multiple assignment: it is not
              allowed to modify a constant ", $1);
};

```

Acciona el nodo *borrar*, que limpia la pantalla.

Listing 30: **Acciones semánticas - clear**

```

clear: CLEAR
{
    // Create a new clear node
    $$ = new lp::ClearStmt();
};

```

Crea el nodo *más-más*, que aumenta en 1 la expresión.

Listing 31: **Acciones semánticas - plusplus**

```

plusplus: PLUSPLUS VARIABLE
{
    // Create a new print node
    $$ = new lp::PlusPlusNode($2);
};

```

Crea el nodo *menos-menos*, que disminuye en 1 la expresión.

Listing 32: **Acciones semánticas - minusminus**

```

minusminus: MINUSMINUS VARIABLE
{
    // Create a new print node
    $$ = new lp::MinusMinusNode($2);
};

```

Acciona el nodo *lugar*, que coloca el cursor del terminal en el lugar seleccionado con las dos expresiones.

Listing 33: **Acciones semánticas - place**

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

Crea el nodo *escribir*, que imprime por pantalla la variable numérica pasada como argumento.

Listing 34: **Acciones semánticas - print**

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

Crea el nodo *leer*, que permite asignar un nuevo valor a una variable numérica.

- Acción por defecto.
- Error por si se intenta editar una constante.

Listing 35: **Acciones semánticas - read**

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

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

Crea el nodo *escribir\_cadena*, muy similar al de *escribir*, que imprime el valor de una variable cadena.

Listing 36: **Acciones semánticas - writestring**

```
writestring: WRITESTRING LPAREN exp RPAREN
{
    // Create a new print node
    $$ = new lp::WriteStringStmt($3);
};
```

Crea el nodo *leer\_cadena*, muy similar al de *leer*, que permite cambiar el valor a una variable cadena.

- Acción por defecto.
- Error por si se intenta editar una constante.

Listing 37: **Acciones semánticas - readstring**

```
readstring: READSTRING LPAREN VARIABLE RPAREN
{
    // Create a new read node
    $$ = new lp::ReadStringStmt($3);
}

| READSTRING LPAREN CONSTANT RPAREN
{
    execerror("Semantic error in \"readstring statement\": it is
        not allowed to modify a constant ", $3);
};
```

Acción semántica que crea nodos de *cadena* y *concatenar*.

Listing 38: **Acciones semánticas - exp (STRING)**

```
exp: STRING
{
    $$ = new lp::StringNode($1);
}

| exp CONCAT exp
{
    $$ = new lp::ConcatNode($1, $3);
};
```

Las acciones semánticas de *exp* crean nodos del AST que permiten manejar operaciones aritméticas y relacionales. Además, aquí también se puede encontrar el funcionamiento de la *listOfExp* con funciones *builtin* que, aunque no se usa en esta práctica, es una buena base para una posible ampliación de las funciones del intérprete más adelante.

Listing 39: **Acciones semánticas - exp (NUMBER)**

```
exp: NUMBER
{
    // Create a new number node
    $$ = new lp::NumberNode($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);
    }

    | exp ENTIRE_DIVISION exp
    {
        // Create a new quotient node
        $$ = new lp::DivisionEnteraNode($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);
}

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

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

| 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);
}

| 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);
}

| exp AND exp
{

```

```

        // Create a new "logic and" node
        $$ = new lp::AndNode($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);
    };

```

Puede utilizar dos acciones semánticas:

- Regla épsilon: crea una lista vacía de expresiones.
- Nueva stmt: añade una nueva expresión a la lista.

#### Listing 40: Acciones semánticas - listOfExp

```

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);
};

```

*restOfListOfExp* contiene dos acciones semánticas que son muy similares a las de *listOfExp*.

#### Listing 41: Acciones semánticas - restOfListOfExp

```

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

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

```

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

## 6 AST

Esta sección está dedicada al árbol de sintaxis abstracta, una representación en forma de árbol de la estructura sintáctica simplificada del código fuente. Veremos las principales modificaciones e implementaciones realizadas sobre las clases *Statement* y *ExpNode*<sup>3</sup>.

---

<sup>3</sup>Dichas implementaciones se han realizado sobre los ficheros fuente “ast.cpp” y “ast.hpp”.

## 6.1 Clase *Statement*

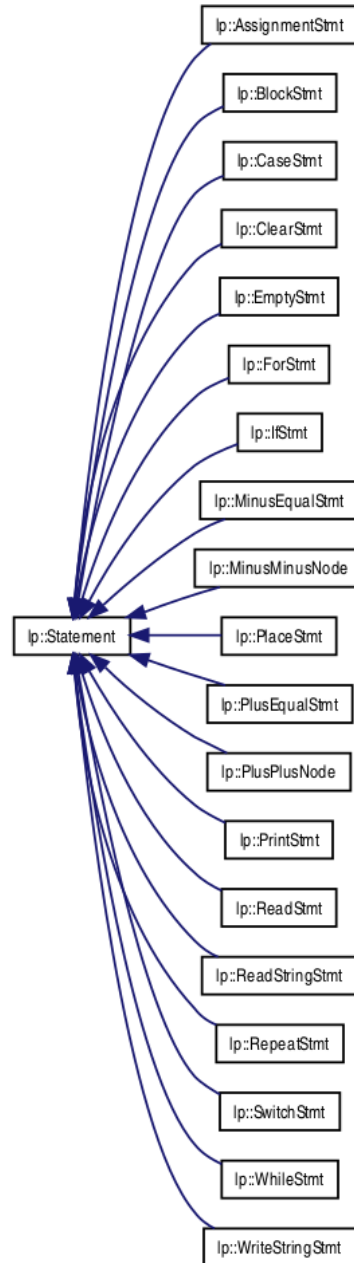


Figura 3: AST - Clase *Statement*.

### 6.1.1 *AssignmentStmt*

Se han añadido las siguientes modificaciones:

- Evaluación de tipos de dato *string* en el método *void lp::AssignmentStmt::evaluate()*:

Listing 42: Método *void lp::AssignmentStmt::evaluate()*

```
...
if (this→_exp != NULL)
{
    ...
    case STRING:
    {
        std::string value;
        // evaluate the expression as STRING
        value = this→_exp→evaluateString();

        if (firstVar→getType() == STRING)
        {
            // Get the identifier in the table of symbols as StringVariable
            lp::StringVariable *v = (lp::StringVariable*)table.getSymbol(this→_id
                );

            // Assignment the value to the identifier in the table of symbols
            v→setValue(value);
        }

        // The type of variable is not STRING

        else
        {
            // Delete the variable from the table of symbols
            table.eraseSymbol(this→_id);

            // Insert the variable in the table of symbols as StringVariable

            // with the type STRING and the value
            lp::StringVariable *v = new lp::StringVariable(this→_id, VARIABLE,
                STRING, value);

            table.installSymbol(v);
        }
    }
    break;
    ...
else
{
    ...
    case STRING:
    {
        /* Get the identifier of the previous asgn in the table of symbols as
            StringVariable */
```

```

lp::StringVariable *secondVar = (lp::StringVariable*)table.getSymbol(this
->_asgn->_id);

// Check the type of the first variable
if (firstVar->getType() == STRING)
{
    /* Get the identifier of the first variable in the table of symbols as
       StringVariable */
    lp::StringVariable *firstVar = (lp::StringVariable*)table.getSymbol(
        this->_id);

    // Assignment the value of the second variable to the first variable
    firstVar->setValue(secondVar->getValue());
}
// The type of variable is not STRING
else
{
    // Delete the first variable from the table of symbols
    table.eraseSymbol(this->_id);

    // Insert the first variable in the table of symbols as StringVariable
    // with the type STRING and the value of the previous variable
    lp::StringVariable *firstVar = new lp::StringVariable(this->_id,
        VARIABLE, STRING, secondVar->getValue());
    table.installSymbol(firstVar);
}
}
break;
...
}

```

### 6.1.2 CaseStmt

Se han implementado las siguientes funcionalidades:

- La propia clase *CaseStmt*, para representar los distintos casos de una sentencia *switch*.
- Variables privadas.
- Constructor.
- Métodos públicos.

Listing 43: Clase *CaseStmt* : *public Statement*

```

/*!
\class    CaseStmt
\brief    Definition of attributes and methods of CaseStmt class
\note     CaseStmt Class publicly inherits from Statement class and adds its own
          print and evaluate functions

```



```

*/

class CaseStmt : public Statement
{
    private:
        ExpNode *_caseCond; ///< Condition of the statement
        std::list<Statement*> *_stmts; ///< Statements

    public:
        ///<
        \brief Constructor of CaseStmt
        \post A new CaseStmt is created
        ///<
        CaseStmt(ExpNode *caseCond, std::list<Statement*> *statement) : _caseCond(
            caseCond), _stmts(statement)
        {
            // Empty
        }

        ///<
        \brief Print the CaseStmt
        \return void
        \sa evaluate()
        ///<
        void print();

        ///<
        \brief Evaluate the CaseStmt
        \return void
        \sa print
        ///<
        void evaluate();

        ///<
        \brief get the condition
        \return ExpNode *
        \sa print
        ///<
        inline ExpNode *getCondition()
        {
            return this->_caseCond;
        }
};

```

Listing 44: Método *void lp::CaseStmt::evaluate()*

```

void lp::CaseStmt::evaluate()
{
    std::list<Statement*>::iterator stmtIter;

    for (stmtIter = this->_stmts->begin(); stmtIter != this->_stmts->end();
        stmtIter++)

```

```
{
    (*stmtIter)->evaluate();
}
}
```

### 6.1.3 *ClearStmt*

Se han implementado las siguientes funcionalidades:

- La propia clase *ClearStmt*, para “limpiar” la pantalla de información.
- Constructor.
- Métodos públicos.

Listing 45: Clase *ClearStmt* : *public Statement*

```
/*!
\class    ClearStmt
\brief    Definition of attributes and methods of ClearStmt class
\note     ClearStmt Class publicly inherits from Statement class and adds its own
          print and evaluate functions
\warning   In this class, print and evaluate functions have the same meaning.
*/
class ClearStmt : public Statement
{
    public:
    /*!
    \brief    Constructor of ClearStmt
    \param expression: pointer to ExpNode
    \post     A new ClearStmt is created with the parameter
    */
    ClearStmt() {}

    /*!
    \brief    Print the ClearStmt
    \return   void
    \sa       evaluate()
    */
    void print();

    /*!
    \brief    Evaluate the ClearStmt
    \return   double
    \sa       print
    */
    void evaluate();
};
```

Listing 46: Método *void lp::ClearStmt::print()*

```
void lp::ClearStmt::print()
{
    std::cout << "Borrar: " << std::endl;
}
```

Listing 47: Método *void lp::ClearStmt::evaluate()*

```
void lp::ClearStmt::evaluate()
{
    std::cout << CLEAR_SCREEN;
}
```

#### 6.1.4 *ForStmt*

Se han realizado las siguientes modificaciones:

- Variables privadas: ahora se usará una lista de *statements*.
- Métodos públicos: *void lp::IfStmt::print()* y *void lp::IfStmt::evaluate()*

Listing 48: Clase *ForStmt : public Statement*

```
/*!
\class    ForStmt
\brief    Definition of attributes and methods of ForStmt class
\note    ForStmt Class publicly inherits from Statement class and adds its own
        print and evaluate functions
*/
class ForStmt : public Statement
{
    private:
        std::string _identifier; //identifier
        ExpNode *_exp1; //1 expresion
        ExpNode *_exp2; //2 expresion
        ExpNode *_stepExp; //step expresion
        std::list<Statement *> *_stmts; //!< Statement of the for loop

    public:
        /*!
        \brief Constructor of ForStmt
        \param condition: ExpNode of the condition
        \param statement: First statement
        \post A new ForStmt is created with the parameters
        */
        ForStmt(std::string identifier, ExpNode *_exp1, ExpNode *_exp2, ExpNode *
            stepExp, std::list<Statement *> *statement) : _identifier(identifier)
        {
            this->_exp1 = exp1;
```

```

        this->_exp2 = exp2;
        this->_stepExp = stepExp;
        this->_stmts = statement;
    }

    ForStmt(std::string identifier, ExpNode *exp1, ExpNode *exp2, std::list<
        Statement *> *statement) : _identifier(identifier)
    {
        this->_exp1 = exp1;
        this->_exp2 = exp2;
        this->_stepExp = NULL;
        this->_stmts = statement;
    }

    /*!
    \brief   Print the ForStmt
    \return  void
    \sa      evaluate
    */
    void print();

    /*!
    \brief   Evaluate the ForStmt
    \return  void
    \sa      print
    */
    void evaluate();
};

```

Listing 49: Método *void lp::ForStmt::print()*

```

void lp::ForStmt::print()
{
    std::cout << "ForStmt: " << std::endl;
    // identifier
    std::cout << this->_identifier;
    // exp1
    this->_exp1->print();
    // exp2
    this->_exp2->print();
    // identifier
    if (this->_stepExp != NULL)
    {
        this->_stepExp->print();
    }

    // Body of the for loop
    std::list<Statement *>::iterator stmtIter;
    for (stmtIter = this->_stmts->begin(); stmtIter != this->_stmts->end();
        stmtIter++)
    {
        std::cout << "\t";
    }
}

```

```

        (*stmtIter)->print();
    }

    std::cout << std::endl;
}

```

Listing 50: Método *void lp::ForStmt::evaluate()*

```

void lp::ForStmt::evaluate()
{
    if (this->_exp1->getType() != NUMBER)
    {
        warning("Runtime error: incompatible types for", "first for expresion");
    }
    else if (this->_exp2->getType() != NUMBER)
    {
        warning("Runtime error: incompatible types for", "second for expresion");
    }
    else
    {
        lp::Variable *var = (lp::Variable *)table.getSymbol(this->_identifier);

        if (var->getType() != NUMBER)
        {
            table.eraseSymbol(this->_identifier);
            lp::NumericVariable *v = new lp::NumericVariable(this->_identifier ,
                VARIABLE, NUMBER, 0);
            table.installSymbol(v);
        }

        int step;
        if (this->_stepExp != NULL)
        {
            step = this->_stepExp->evaluateNumber();
        }
        else
        {
            step = 1;
        }

        if (step == 0)
        {
            warning("Infinite loop in for: ", "step = zero");
        }
        else if ((this->_exp1->evaluateNumber() > this->_exp2->evaluateNumber())
            and step > 0)
        {
            warning("Infinite loop in for: ", "'from' is > than 'to' and step is +
                ");
        }
        else if ((this->_exp1->evaluateNumber() < this->_exp2->evaluateNumber())
            and step < 0)

```

```

{
    warning("Infinite loop in for: ", "'to' is > than 'from' and step is -
        ");
}
else
{
    lp::NumericVariable *v = (lp::NumericVariable *)table.getSymbol(this->
        _identifier);

    for (int a = this->_exp1->evaluateNumber(); a <= this->_exp2->
        evaluateNumber(); a = a + step)
    {
        v->setValue(a);
        std::list<Statement *>::iterator stmtIter;

        for (stmtIter = this->_stmts->begin(); stmtIter != this->_stmts->
            end(); stmtIter++)
        {
            (*stmtIter)->evaluate();
        }
    }
}
}

```

### 6.1.5 *IfStmt*

Se han realizado las siguientes modificaciones:

- Variables privadas: ahora se usarán listas para guardar tanto antecedentes como consecuentes.
- Métodos públicos: `void lp::IfStmt::print()` y `void lp::IfStmt::evaluate()`

Listing 51: **Clase** *IfStmt* : *public Statement*

```

/*!
  \class    IfStmt
  \brief    Definition of attributes and methods of IfStmt class
  \note     IfStmt Class publicly inherits from Statement class
            and adds its own print and evaluate functions
*/
class IfStmt : public Statement
{
    private:
        ExpNode *_cond; //!< Condicion of the if statement
        std::list<Statement *> *_stmt1; //!< Statement of the consequent
        std::list<Statement *> *_stmt2; //!< Statement of the alternative

    public:

```

```

    /*!
    \brief Constructor of Single IfStmt (without alternative)
    \param condition: ExpNode of the condition
    \param statement1: Statement of the consequent
    \post A new IfStmt is created with the parameters
    */
    IfStmt(ExpNode *condition, std::list<Statement *> *statement1)
    {
        this->_cond = condition;
        this->_stmt1 = statement1;
    }

    /*!
    \brief Constructor of Compound IfStmt (with alternative)
    \param condition: ExpNode of the condition
    \param statement1: Statement of the consequent
    \param statement2: Statement of the alternative
    \post A new IfStmt is created with the parameters
    */
    IfStmt(ExpNode *condition, std::list<Statement *> *statement1, std::list<
        Statement *> *statement2)
    {
        this->_cond = condition;
        this->_stmt1 = statement1;
        this->_stmt2 = statement2;
    }

    IfStmt(ConstantNode *condition, std::list<Statement *> *statement1)
    {
        this->_cond = (ExpNode *)condition;
        this->_stmt1 = statement1;
    }

    IfStmt(ConstantNode *condition, std::list<Statement *> *statement1, std::
        list<Statement *> *statement2)
    {
        this->_cond = (ExpNode *)condition;
        this->_stmt1 = statement1;
        this->_stmt2 = statement2;
    }

    /*!
    \brief Print the IfStmt
    \return void
    \sa evaluate
    */
    void print();

    /*!
    \brief Evaluate the IfStmt
    \return void
    \sa print
    */

```

```

    void evaluate();
};

```

Listing 52: Método *void lp::IfStmt::print()*

```

void lp::IfStmt::print()
{
    std::cout << "IfStmt: " << std::endl;
    // Condition
    std::cout << "\t";
    this->_cond->print();

    // Consequent
    std::list<Statement*>::iterator stmtIter;
    for (stmtIter = this->_stmt1->begin(); stmtIter != this->_stmt1->end();
        stmtIter++)
        (*stmtIter)->print();

    // The alternative is printed if exists
    for (stmtIter = this->_stmt2->begin(); stmtIter != this->_stmt2->end();
        stmtIter++)
        (*stmtIter)->print();

    std::cout << std::endl;
}

```

Listing 53: Método *void lp::IfStmt::evaluate()*

```

void lp::IfStmt::evaluate()
{
    //check if the condition is boolean
    if (this->_cond->getType() != BOOL)
    {
        warning("Runtime error: incompatible types for", "if condition");
    }
    else
    {
        // If the condition is true,
        if (this->_cond->evaluateBool() == true)
        {
            // the consequent is run
            std::list<Statement*>::iterator stmtIter;

            for (stmtIter = this->_stmt1->begin(); stmtIter != this->
                _stmt1->end(); stmtIter++)
                (*stmtIter)->evaluate();
        }

        // Otherwise, the alternative is run if exists
        else if (this->_stmt2 != NULL)

```



```

    {
        std::list<Statement *>::iterator stmtIter;

        for (stmtIter = this→_stmt2→begin(); stmtIter != this→
            _stmt2→end(); stmtIter++)
            (*stmtIter)→evaluate();
    }
}

```

### 6.1.6 *MinusEqualStmt*

Se han implementado las siguientes funcionalidades:

- La propia clase, para representar el operador de asignación -=.
- Variables privadas.
- Métodos públicos: *void lp::MinusEqualStmt::print()* y *void lp::MinusEqualStmt::evaluate()*

Listing 54: Class *MinusEqualStmt* : *public Statement*

```

/*!
\class      MinusEqualStmt
\brief      Definition of attributes and methods of MinusEqualStmt class
\note      MinusEqualStmt Class publicly inherits from Statement class and adds
            its own print and evaluate functions
*/

class MinusEqualStmt : public Statement
{
private:
    std::string _id; //!< Name of the variable of the assignment
                      statement
    ExpNode *_exp;   //!< Expression the assignment statement

public:
/*!
\brief      Constructor of MinusEqualStmt
\param id: string, variable of the MinusEqualStmt
\param expression: pointer to ExpNode
\param post A new MinusEqualStmt is created with the parameters
*/
    MinusEqualStmt(std::string id, ExpNode *expression) : _id(id),
        _exp(expression)
    {
    }

/*!
\brief      Print the MinusEqualStmt
\return     void
\sa         evaluate()
*/

```

```

*/
        void print();

/*!
    \brief    Evaluate the MinusEqualStmt
    \return   void
    \sa       print
*/
        void evaluate();
};

```

Listing 55: Método *void lp::MinusEqualStmt::print()*

```

void lp::MinusEqualStmt::print()
{
    std::cout << "minus_equal_node: -:=" << std::endl;
    std::cout << "\t";
    std::cout << this->_id << std::endl;
    std::cout << "\t";

    // Check the expression
    if (this->_exp != NULL)
    {
        this->_exp->print();
        std::cout << std::endl;
    }
}

```

Listing 56: Método *void lp::MinusEqualStmt::evaluate()*

```

void lp::MinusEqualStmt::evaluate()
{
    /* Get the identifier in the table of symbols as Variable */
    lp::Variable *firstVar = (lp::Variable *)table.getSymbol(this->_id);
    lp::NumericVariable *var = (lp::NumericVariable *)table.getSymbol(this->
        _id);
    // Check the expression
    if (this->_exp != NULL)
    {
        double value;
        // evaluate the expression as NUMBER
        value = this->_exp->evaluateNumber();

        // Check the type of the first variable
        if (firstVar->getType() == NUMBER)
        {
            // Get the identifier in the table of symbols as
            NumericVariable
            lp::NumericVariable *v = (lp::NumericVariable *)table.
                getSymbol(this->_id);

```

```

        // Assignment the value to the identifier in the table of
        // symbols
        v->setValue(var->getValue() - value);
    }
    // The type of variable is not NUMBER
    else
    {
        // Delete the variable from the table of symbols
        table.eraseSymbol(this->_id);

        // Insert the variable in the table of symbols as
        // NumericVariable
        // with the type NUMBER and the value
        lp::NumericVariable *v = new lp::NumericVariable(this->_id
            , VARIABLE, NUMBER, var->getValue() - value);
        table.installSymbol(v);
    }
}

```

### 6.1.7 MinusMinusNode

Se han implementado las siguientes funcionalidades:

- La propia clase, para representar el operador - -.
- Variables privadas.
- Métodos públicos: *void lp::MinusMinusNode::print()* y *void lp::MinusMinusNode::evaluate()*

Listing 57: Clase *MinusMinusNode* : *public Statement*

```

/*!
\class    MinusMinusNode
\brief    Definition of attributes and methods of MinusMinusNode class
\note    MinusMinusNode Class publicly inherits from Statement class
         and adds its own print and evaluate functions
*/

class MinusMinusNode : public Statement
{
private:
    std::string _id; //!< Name of the variable of the minus minus
                    statement

public:
/*!
\brief    Constructor of MinusMinusNode
\param id: string, variable of the MinusMinusNode
\param expression: pointer to ExpNode
\post    A new MinusMinusNode is created with the parameters
*/

```

```

        MinusMinusNode(std::string id)
        {
            this->_id = id;
        }

    /*!
        \brief    Print the MinusMinusNode
        \return   void
        \sa       evaluate()
    */
    void print();

    /*!
        \brief    Evaluate the MinusMinusNode
        \return   void
        \sa       print
    */
    void evaluate();
};

```

Listing 58: Método *void lp::MinusMinusNode::print()*

```

void lp::MinusMinusNode::print()
{
    std::cout << "minus_minus_node: —" << std::endl;
    std::cout << "\t";
    std::cout << this->_id << std::endl;
}

```

Listing 59: Método *void lp::MinusMinusNode::evaluate()*

```

void lp::MinusMinusNode::evaluate()
{
    lp::Variable *firstVar = (lp::Variable *)table.getSymbol(this->_id);
    lp::NumericVariable *var = (lp::NumericVariable *)table.getSymbol(this->
        _id);

    if (firstVar->getType() == NUMBER)
    {
        lp::NumericVariable *v = (lp::NumericVariable *)table.getSymbol(
            this->_id);
        v->setValue(var->getValue() - 1);
    }
}

```

### 6.1.8 PlaceStmt

Se han implementado las siguientes funcionalidades:

- La propia clase, para colocar el cursor en una posición específica.
- Variables privadas.
- Métodos públicos: *void lp::PlaceStmt::print()* y *void lp::PlaceStmt::evaluate()*

Listing 60: Clase *PlaceStmt* : *public Statement*

```

/*!
    \class    PlaceStmt
    \brief    Definition of attributes and methods of PlaceStmt class
    \note     PlaceStmt Class publicly inherits from Statement class
              and adds its own print and evaluate functions
    \warning  In this class, print and evaluate functions have the same meaning.
*/

    class PlaceStmt : public Statement
    {
    private:
        ExpNode *_left;
        ExpNode *_right;

    public:

/*!
        \brief Constructor of ClearStmt
        \param expression: pointer to ExpNode
        \post A new PlaceStmt is created with the parameter
*/

        PlaceStmt(ExpNode *left, ExpNode *right)
        {
            this->_left = left;
            this->_right = right;
        }

/*!
        \brief Print the ClearStmt
        \return void
        \sa      evaluate()
*/

        void print();

/*!
        \brief Evaluate the ClearStmt
        \return double
        \sa      print
*/

        void evaluate();
    };

```

Listing 61: Método *void lp::PlaceStmt::print()*

```

void lp::PlaceStmt::print()
{

```

```
std::cout << "Lugar: " << std::endl;
this->_left->print();
this->_right->print();
}
```

Listing 62: Método *void lp::PlaceStmt::evaluate()*

```
void lp::PlaceStmt::evaluate()
{
    PLACE((int)this->_left->evaluateNumber(), (int)this->_right->
        evaluateNumber());
}
```

### 6.1.9 *PlusEqualStmt*

Se han implementado las siguientes funcionalidades:

- La propia clase, para representar el operador de asignación +=.
- Variables privadas.
- Métodos públicos: *void lp::PlusEqualStmt::print()* y *void lp::PlusEqualStmt::evaluate()*

Listing 63: Clase *PlusEqualStmt : public Statement*

```
/*!
\class    PlusEqualStmt
\brief    Definition of attributes and methods of PlusEqualStmt class
\note     PlusEqualStmt Class publicly inherits from Statement class
          and adds its own print and evaluate functions
*/

class PlusEqualStmt : public Statement
{
private:
    std::string _id; //!< Name of the variable of the assignment
                    statement
    ExpNode *_exp;  //!< Expression the assignment statement

public:

/*!
\brief    Constructor of PlusEqualStmt
\param id: string, variable of the PlusEqualStmt
\param expression: pointer to ExpNode
\post     A new PlusEqualStmt is created with the parameters
*/

    PlusEqualStmt(std::string id, ExpNode *expression) : _id(id), _exp
        (expression)
    {
    }
}
```

```

/*!
    \brief    Print the PlusEqualStmt
    \return   void
    \sa       evaluate()
*/
    void print();

/*!
    \brief    Evaluate the PlusEqualStmt
    \return   void
    \sa       print
*/
    void evaluate();
};

```

Listing 64: Método *void lp::PlusEqualStmt::print()*

```

void lp::PlusEqualStmt::print()
{
    std::cout << "plus_equal_node: +:=" << std::endl;
    std::cout << "\t";
    std::cout << this->_id << std::endl;
    std::cout << "\t";

    // Check the expression
    if (this->_exp != NULL)
    {
        this->_exp->print();
        std::cout << std::endl;
    }
}

```

Listing 65: Método *void lp::PlusEqualStmt::evaluate()*

```

void lp::PlusEqualStmt::evaluate()
{
    /* Get the identifier in the table of symbols as Variable */
    lp::Variable *firstVar = (lp::Variable *)table.getSymbol(this->_id);
    lp::NumericVariable *var = (lp::NumericVariable *)table.getSymbol(this->
        _id);
    // Check the expression
    if (this->_exp != NULL)
    {
        double value;
        // evaluate the expression as NUMBER
        value = this->_exp->evaluateNumber();

        // Check the type of the first variable
        if (firstVar->getType() == NUMBER)
        {

```

```

        // Get the identifier in the table of symbols as
        NumericVariable
        lp::NumericVariable *v = (lp::NumericVariable *)table.
            getSymbol(this→_id);

        // Assignment the value to the identifier in the table of
        symbols
        v→setValue(value + var→getValue());
    }
    // The type of variable is not NUMBER
    else
    {
        // Delete the variable from the table of symbols
        table.eraseSymbol(this→_id);

        // Insert the variable in the table of symbols as
        NumericVariable
        // with the type NUMBER and the value
        lp::NumericVariable *v = new lp::NumericVariable(this→_id
            , VARIABLE, NUMBER, value + var→getValue());
        table.installSymbol(v);
    }
}

```

#### 6.1.10 *PlusPlusNode*

Se han implementado las siguientes funcionalidades:

- La propia clase, para representar el operador + +.
- Variables privadas.
- Métodos públicos: *void lp::PlusPlusNode::print()* y *void lp::PlusPlusNode::evaluate()*

Listing 66: Clase *PlusPlusNode* : *public Statement*

```

/*!
 \class    PlusPlusNode
 \brief    Definition of attributes and methods of PlusPlusNode class
 \note     PlusPlusNode Class publicly inherits from Statement class
           and adds its own print and evaluate functions
*/

class PlusPlusNode : public Statement
{
private:
    std::string _id; //!< Name of the variable of the plusplus
                    statement

public:
}
/*!

```



```

    \brief Constructor of PlusPlusNode
    \param id: string, variable of the PlusPlusNode
    \param expression: pointer to ExpNode
    \post A new PlusPlusNode is created with the parameters
*/
    PlusPlusNode(std::string id)
    {
        this->_id = id;
    }

/*!
    \brief Print the PlusPlusNode
    \return void
    \sa evaluate()
*/
    void print();

/*!
    \brief Evaluate the PlusPlusNode
    \return void
    \sa print
*/
    void evaluate();
};

```

Listing 67: Método *void lp::PlusPlusNode::print()*

```

void lp::PlusEqualStmt::print()
{
    std::cout << "plus_equal_node: +:=" << std::endl;
    std::cout << "\t";
    std::cout << this->_id << std::endl;
    std::cout << "\t";

    // Check the expression
    if (this->_exp != NULL)
    {
        this->_exp->print();
        std::cout << std::endl;
    }
}

```

Listing 68: Método *void lp::PlusPlusNode::evaluate()*

```

void lp::PlusEqualStmt::evaluate()
{
    /* Get the identifier in the table of symbols as Variable */
    lp::Variable *firstVar = (lp::Variable *)table.getSymbol(this->_id);
    lp::NumericVariable *var = (lp::NumericVariable *)table.getSymbol(this->
        _id);
    // Check the expression

```

```

if (this→_exp != NULL)
{
    double value;
    // evaluate the expression as NUMBER
    value = this→_exp→evaluateNumber();

    // Check the type of the first variable
    if (firstVar→getType() == NUMBER)
    {
        // Get the identifier in the table of symbols as
        // NumericVariable
        lp::NumericVariable *v = (lp::NumericVariable *)table.
            getSymbol(this→_id);

        // Assignment the value to the identifier in the table of
        // symbols
        v→setValue(value + var→getValue());
    }
    // The type of variable is not NUMBER
    else
    {
        // Delete the variable from the table of symbols
        table.eraseSymbol(this→_id);

        // Insert the variable in the table of symbols as
        // NumericVariable
        // with the type NUMBER and the value
        lp::NumericVariable *v = new lp::NumericVariable(this→_id
            , VARIABLE, NUMBER, value + var→getValue());
        table.installSymbol(v);
    }
}

```

#### 6.1.11 *ReadStringStmt*

Se han implementado las siguientes funcionalidades:

- La propia clase, para efectuar la lectura de cadenas alfanuméricas.
- Variables privadas.
- Métodos públicos: *void lp::ReadStringStmt::print()* y *void lp::ReadStringStmt::evaluate()*

Listing 69: Clase *ReadStringStmt* : *public Statement*

```

/*!
\class    ReadStringStmt
\brief    Definition of attributes and methods of ReadStringStmt class
\note    ReadStringStmt Class publicly inherits from Statement class
          and adds its own print and evaluate functions

```

```

*/
class ReadStringStmt : public Statement
{
private:
    std::string _id; ///< Name of the ReadStringStmt

public:
    ///
    ReadStringStmt(std::string id)
    {
        this->_id = id;
    }

    ///
    void print();

    ///
    void evaluate();
};

```

Listing 70: Método *void lp::ReadStringStmt::print()*

```

void lp::ReadStringStmt::print()
{
    std::cout << "ReadStringStmt: " << std::endl;
    std::cout << " readstring (" << this->_id << ")" ;
    std::cout << std::endl;
}

```

Listing 71: Método *void lp::ReadStringStmt::evaluate()*

```

void lp::ReadStringStmt::evaluate()
{
    std::string tempString;
    std::cin >> tempString;

    ///< Get the identifier in the table of symbols as Variable */
    lp::Variable *var = (lp::Variable *)table.getSymbol(this->_id);
}

```

```

// Check if the type of the variable is NUMBER
if (var->getType() == STRING)
{
    /* Get the identifier in the table of symbols as NumericVariable
    */
    lp::StringVariable *n = (lp::StringVariable *)table.getSymbol(this
->-id);

    /* Assignment the read value to the identifier */
    n->setValue(tempString);
}
// The type of variable is not NUMBER
else
{
    // Delete $1 from the table of symbols as Variable
    table.eraseSymbol(this->-id);

    // Insert $1 in the table of symbols as NumericVariable
    // with the type NUMBER and the read value
    lp::StringVariable *n = new lp::StringVariable(this->-id, VARIABLE
, STRING, tempString);

    table.installSymbol(n);
}
}

```

### 6.1.12 RepeatStmt

Se han implementado las siguientes funcionalidades:

- La propia clase, para implementar la funcionalidad de una sentencia *do*.
- Variables privadas.
- Métodos públicos: *void lp::RepeatStmt::print()* y *void lp::RepeatStmt::evaluate()*

Listing 72: Clase *RepeatStmt* : *public Statement*

```

/*!
\class    RepeatStmt
\brief    Definition of attributes and methods of RepeatStmt class
\note    RepeatStmt Class publicly inherits from Statement class
          and adds its own print and evaluate functions
*/

class RepeatStmt : public Statement
{
private:
    ExpNode *_cond;                                //!< Condicion of
                                                    the repeat statement
    std::list<Statement *> *_stmts;                //!< Statement of the body of the
                                                    repeat loop

```

```

public:

/*!
    \brief Constructor of RepeatStmt
    \param condition: ExpNode of the condition
    \param statement: Statement of the body of the loop
    \post A new RepeatStmt is created with the parameters
*/
    RepeatStmt(std::list<Statement*> *statement, ExpNode *condition)
    {
        this->_stmts = statement;
        this->_cond = condition;
    }

/*!
    \brief Print the RepeatStmt
    \return void
    \sa evaluate
*/
    void print();

/*!
    \brief Evaluate the RepeatStmt
    \return void
    \sa print
*/
    void evaluate();
};

```

Listing 73: Método *void lp::RepeatStmt::print()*

```

void lp::RepeatStmt::print()
{
    std::cout << "RepeatStmt: " << std::endl;
    // Condition
    std::cout << "\t";
    this->_cond->print();

    // Body of the repeat loop
    std::list<Statement*>::iterator stmtIter;

    for (stmtIter = this->_stmts->begin(); stmtIter != this->_stmts->end();
        stmtIter++)
        (*stmtIter)->print();

    std::cout << std::endl;
}

```

Listing 74: Método *void lp::RepeatStmt::evaluate()*

```

void lp::RepeatStmt::evaluate()

```

```

{
    // While the condition is true. the body is run
    do
    {
        std::list<Statement *>::iterator stmtIter;

        for (stmtIter = this→_stmts→begin(); stmtIter != this→_stmts→
            end(); stmtIter++)
            (*stmtIter)→evaluate();

    } while (this→_cond→evaluateBool() == false);
}

```

### 6.1.13 *SwitchStmt*

Se han implementado las siguientes funcionalidades:

- La propia clase, para implementar la funcionalidad de una sentencia *switch*.
- Variables privadas.
- Métodos públicos: *void lp::SwitchStmt::print()* y *void lp::SwitchStmt::evaluate()*

Listing 75: Clase *SwitchStmt* : *public Statement*

```

/*!
\class      SwitchStmt
\brief      Definition of attributes and methods of SwitchStmt class
\note      SwitchStmt Class publicly inherits from Statement class
            and adds its own print and evaluate functions
*/

class SwitchStmt : public Statement
{
private:
    ExpNode *_exp;
    std::list<CaseStmt *> *_stmts1;    //!< Cases
    std::list<Statement *> *_stmts2;  //!< Default

public:
/*!
\brief      Constructor of SwitchStmt
\param condition: ExpNode of the condition
\param statement: First statement
\post      A new SwitchStmt is created with the parameters
*/

    SwitchStmt(ExpNode *exp, std::list<CaseStmt *> *stmts1, std::list<
        Statement *> *stmts2 = NULL)
    {
        this->_exp = exp;
        this->_stmts1 = stmts1;
        this->_stmts2 = stmts2;
    }
};

```

```

    }

    /*!
    \brief    Print the SwitchStmt
    \return   void
    \sa      evaluate
    */

    void print();

    /*!
    \brief    Evaluate the SwitchStmt
    \return   void
    \sa      print
    */

    void evaluate();
};

```

Listing 76: Método *void lp::SwitchStmt::evaluate()*

```

void lp::SwitchStmt::evaluate()
{
    bool def = true;

    if (_stmts2 == NULL)
    {
        def = false;
    }

    switch (this->_exp->getType())
    {
    case NUMBER:

        if (_stmts1->size() != 0)
        {
            std::list<CaseStmt*>::iterator caseIt;
            double variable = _exp->evaluateNumber();

            for (caseIt = this->_stmts1->begin(); caseIt != this->
                _stmts1->end(); caseIt++)
            {
                if (std::abs((variable - (*caseIt)->getCondition()
                    ->evaluateNumber())) < ERROR.BOUND)
                {
                    (*caseIt)->evaluate();
                    def = false;
                }
            }
        }
        if (def)
        {
            std::list<Statement*>::iterator stmtIt;

```

```

        for (stmtIt = this->_stmts2->begin(); stmtIt != this->
            _stmts2->end(); stmtIt++)
        {
            (*stmtIt)->evaluate();
        }

        def = false;
    }
    break;

case BOOL:

    if (_stmts1->size() != 0)
    {
        std::list<CaseStmt *>::iterator caseIt;
        bool variable = _exp->evaluateBool();

        for (caseIt = this->_stmts1->begin(); caseIt != this->
            _stmts1->end(); caseIt++)
        {
            if (variable == (*caseIt)->getCondition()->
                evaluateBool())
            {
                (*caseIt)->evaluate();
                def = false;
            }
        }
    }
    if (def)
    {
        std::list<Statement *>::iterator stmtIt;

        for (stmtIt = this->_stmts2->begin(); stmtIt != this->
            _stmts2->end(); stmtIt++)
        {
            (*stmtIt)->evaluate();
        }

        def = false;
    }
    break;

case STRING:

    if (_stmts1->size() != 0)
    {
        std::list<CaseStmt *>::iterator caseIt;
        std::string variable = _exp->evaluateString();

        for (caseIt = this->_stmts1->begin(); caseIt != this->
            _stmts1->end(); caseIt++)
        {
            if (variable == (*caseIt)->getCondition()->

```



```

        evaluateString()
    {
        (*caseIt)->evaluate();
        def = false;
    }
}
if (def)
{
    std::list<Statement *>::iterator stmtIt;

    for (stmtIt = this->_stmts2->begin(); stmtIt != this->
        _stmts2->end(); stmtIt++)
    {
        (*stmtIt)->evaluate();
    }

    def = false;
}
break;

default:
    warning("Runtime error: incompatible types of parameters for ", "
        Switch statement");
}
}

```

#### 6.1.14 *WhileStmt*

Se han implementado las siguientes funcionalidades:

- La propia clase, para implementar la funcionalidad de una sentencia de bucle *while*.
- Variables privadas.
- Métodos públicos: *void lp::WhileStmt::print()* y *void lp::WhileStmt::evaluate()*

Listing 77: Clase *WhileStmt* : *public Statement*

```

/*!
\class    WhileStmt
\brief    Definition of attributes and methods of WhileStmt class
\note    WhileStmt Class publicly inherits from Statement class
            and adds its own print and evaluate functions
*/

class WhileStmt : public Statement
{
private:
    ExpNode *_cond;                                //!< Condicion of
            the while statement

```

```

        std::list<Statement*> *_stmts; /// Statement of the body of the
            while loop

    public:
/*!
    \brief Constructor of WhileStmt
    \param condition: ExpNode of the condition
    \param statement: Statement of the body of the loop
    \post A new WhileStmt is created with the parameters
*/

        WhileStmt(ExpNode *condition, std::list<Statement*> *statement)
        {
            this->_cond = condition;
            this->_stmts = statement;
        }

/*!
    \brief Print the WhileStmt
    \return void
    \sa evaluate
*/

        void print();

/*!
    \brief Evaluate the WhileStmt
    \return void
    \sa print
*/

        void evaluate();
};

```

Listing 78: Método *void lp::WhileStmt::print()*

```

void lp::WhileStmt::print()
{
    std::cout << "WhileStmt: " << std::endl;
    // Condition
    std::cout << "\t";
    this->_cond->print();

    // Body of the while loop
    std::list<Statement*>::iterator stmtIter;

    for (stmtIter = this->_stmts->begin(); stmtIter != this->_stmts->end();
        stmtIter++)
    {
        std::cout << "\t";
        (*stmtIter)->print();
    }

    std::cout << std::endl;
}

```

Listing 79: Método *void lp::WhileStmt::evaluate()*

```

void lp::WhileStmt::evaluate()
{
    // While the condition is true. the body is run
    while (this->_cond->evaluateBool() == true)
    {
        std::list<Statement*>::iterator stmtIter;

        for (stmtIter = this->_stmts->begin(); stmtIter != this->_stmts->
            end(); stmtIter++)
        {
            (*stmtIter)->evaluate();
        }
    }
}

```

### 6.1.15 WriteStringStmt

Se han implementado las siguientes funcionalidades:

- La propia clase, para efectuar la escritura de cadenas alfanuméricas.
- Variables privadas.
- Métodos públicos: *void lp::WriteStringStmt::print()* y *void lp::WriteStringStmt::evaluate()*

Listing 80: Clase *WriteStringStmt : public Statement*

```

/*!
\class    WriteStringStmt
\brief    Definition of attributes and methods of WriteStringStmt class
\note    WriteStringStmt Class publicly inherits from Statement class
          and adds its own print and evaluate functions
\warning  In this class, print and evaluate functions have the same meaning.
*/

class WriteStringStmt : public Statement
{
private:
    ExpNode *_exp; //!< Expression the print statement

public:
/*!
\brief    Constructor of WriteStringStmt
\param expression: pointer to ExpNode
\post    A new WriteStringStmt is created with the parameter
*/

    WriteStringStmt(ExpNode *expression)
    {
        this->_exp = expression;
    }
}

```

```

/*!
    \brief    Print the WriteStringStmt
    \return   void
    \sa       evaluate()
*/
    void print();

/*!
    \brief    Evaluate the WriteStringStmt
    \return   double
    \sa       print
*/
    void evaluate();
};

```

Listing 81: Método *void lp::WriteStringStmt::print()*

```

void lp::WriteStringStmt::print()
{
    std::cout << "WriteStringStmt: " << std::endl;
    std::cout << " writeString ";
    this->_exp->print();
    std::cout << std::endl;
}

```

Listing 82: Método *void lp::WriteStringStmt::evaluate()*

```

void lp::WriteStringStmt::evaluate()
{
    if (this->_exp->getType() == STRING)
    {
        std::string aux = this->_exp->evaluateString();
        for (unsigned i = 0; i < aux.size(); i++)
        {
            if (aux[i] != '\\')
            {
                std::cout << aux[i];
            }
            else
            {
                i++;
                switch (aux[i])
                {
                    case 't':
                        std::cout << "\t";
                        break;
                    case 'n':
                        std::cout << "\n";
                        break;

```

```
        case '\\':
            std::cout << "'";
            break;
        default:
            std::cout << "\\\" << aux[i];
        }
    }
}
else
{
    warning("Runtime error: incompatible type for ", "writeString");
}
}
```

## 6.2 Clase *ExpNode*

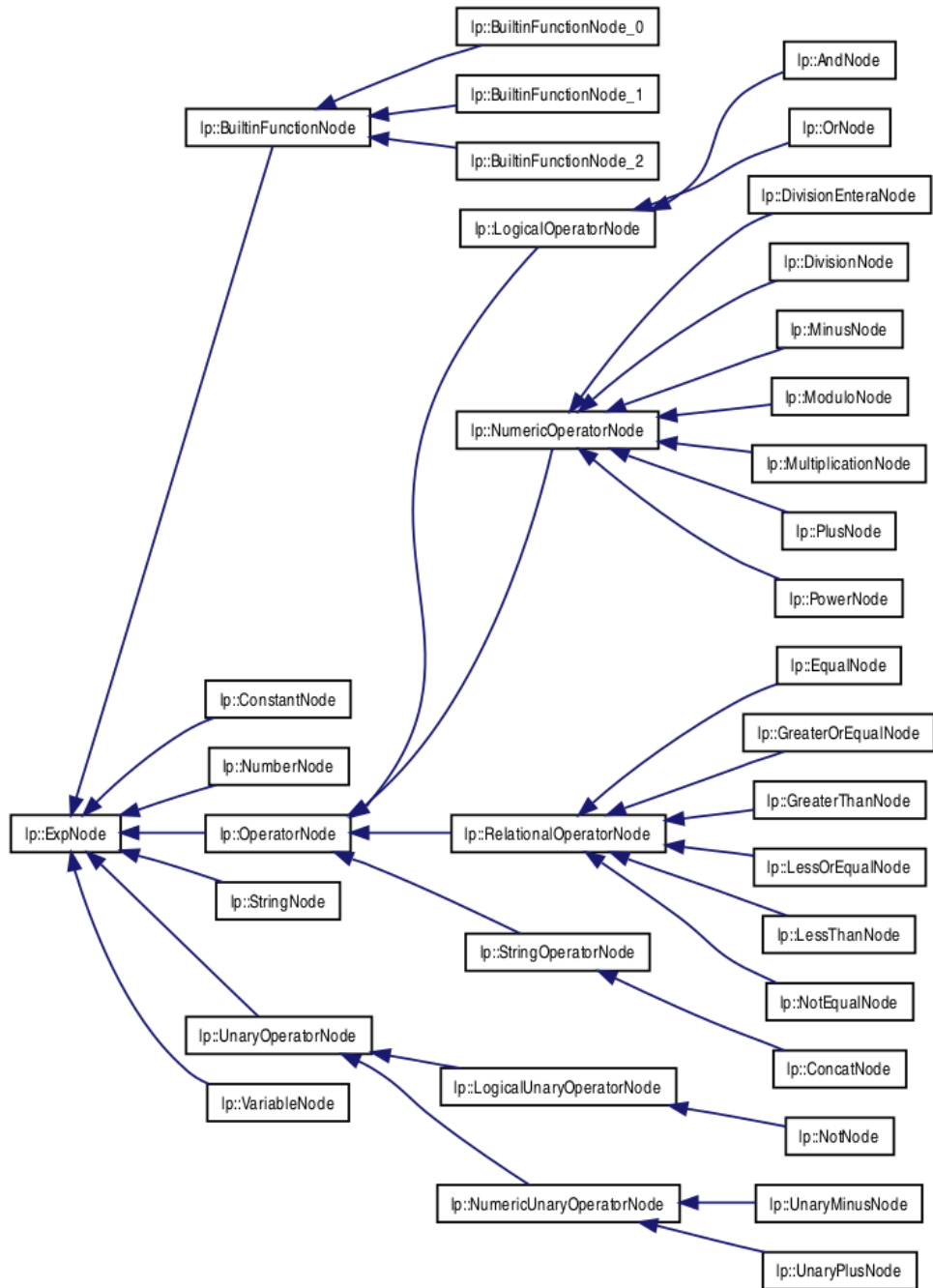


Figura 4: AST - Clase *ExpNode*.

### 6.2.1 *DivisionEnteraNode*

Se han implementado las siguientes funcionalidades:

- La propia clase, para efectuar la división entera de dos elementos numéricos.
- Métodos públicos: *void lp::DivisionEnteraNode::print()* y *void lp::DivisionEnteraNode::evaluate()*

Listing 83: Clase ***DivisionEnteraNode : public NumericOperatorNode***

```
class DivisionEnteraNode : public NumericOperatorNode
{
    public:
        /*!
        \brief Constructor of DivisionNode uses NumericOperatorNode's
            constructor as members initializer
        \param L: pointer to ExpNode
        \param R: pointer to ExpNode
        \post A new DivisionEnteraNode is created with the parameter
        */
        DivisionEnteraNode(ExpNode *L, ExpNode *R) : NumericOperatorNode(L
            , R)
        {
            // Empty
        }
        /*!
        \brief Print the DivisionEnteraNode
        \return void
        \sa evaluate()
        */
        void print();

        /*!
        \brief Evaluate the DivisionEnteraNode
        \return double
        \sa print
        */
        double evaluateNumber();
};
```

Listing 84: Método ***void lp::DivisionEnteraNode::print()***

```
void lp::DivisionEnteraNode::print()
{
    std::cout << "DivisionEnteraNode: " << std::endl;
    this->_left->print();
    std::cout << " #div ";
    this->_right->print();
}
```

Listing 85: Método ***void lp::DivisionEnteraNode::evaluate()***

```
double lp::DivisionEnteraNode::evaluateNumber()
```

```

{
    int result = 0.0;

    // Ckeck the types of the expressions
    if (this→getType() == NUMBER)
    {
        double leftNumber, rightNumber;

        leftNumber = this→_left →evaluateNumber();
        rightNumber = this→_right →evaluateNumber();

        // The divisor is not zero
        if (std::abs(rightNumber) > ERROR_BOUND)
        {
            result = leftNumber / rightNumber;
        }
        else
        {
            warning("Runtime error", "Division by zero");
        }
    }
    else
    {
        warning("Runtime error: the expressions are not numeric for", "
            Entire division");
    }

    return result;
}

```

### 6.2.2 ConcatNode

Se han implementado las siguientes funcionalidades:

- La propia clase, para efectuar la concatenación de dos variables alfanuméricas.
- Métodos públicos: *void lp::Concat::print()* y *void lp::Concat::evaluate()*

Listing 86: Clase *Concat* : *public StringOperatorNode*

```

/*!
    \class    ConcatNode
    \brief    Definition of attributes and methods of ConcatNode class
    \note     ConcatNode Class publicly inherits from NumericOperatorNode class
              and adds its own print and evaluate functions
*/

class ConcatNode : public StringOperatorNode
{
    public:
}
/*!

```



```

    \brief Constructor of ConcatNode uses NumericOperatorNode's
           constructor as members initializer
    \param L: pointer to ExpNode
    \param R: pointer to ExpNode
    \post A new ConcatNode is created with the parameter

*/

ConcatNode(ExpNode *L, ExpNode *R) : StringOperatorNode(L, R)
{
    // Empty
}

/*!
 \brief   Print the ConcatNode
 \return  void
 \sa      evaluate()
*/

void print();

/*!
 \brief   Evaluate the ConcatNode
 \return  string
 \sa      print
*/

std::string evaluateString();
};

```

Listing 87: Método *void lp::Concat::print()*

```

void lp::ConcatNode::print()
{
    std::cout << "ConcatNode: " << std::endl;
    this->_left->print();
    std::cout << " || ";
    this->_right->print();
}

```

Listing 88: Método *void lp::Concat::evaluate()*

```

std::string lp::ConcatNode::evaluateString()
{
    std::string result = "";

    // Ckeck the types of the expressions
    if (this->getType() == STRING)
    {
        result = this->_left->evaluateString() + this->_right->
            evaluateString();
    }
    else
    {
        warning("Runtime error: the expressions are not string for", "
            concatenation");
    }
}

```

```

        return result;
    }

```

### 6.2.3 *StringNode*

Se han implementado las siguientes funcionalidades:

- La propia clase.
- Métodos públicos:
  - *void lp::StringNode::type()*.
  - *void lp::StringNode::print()*.
  - *void lp::StringNode::evaluate()*.

Listing 89: Clase *StringNode* : *public ExpNode*

```

/*!
 \class StringNode
 \brief Definition of attributes and methods of StringNode class
 \note StringNode Class publicly inherits from ExpNode class
*/

class StringNode : public ExpNode
{
private:
    std::string _string; //!< \brief number of the StringNode

public:
/*!
 \brief Constructor of StringNode
 \param value: std::string
 \post A new StringNode is created with the value of the parameter
 \note Inline function
*/
    StringNode(std::string value)
    {
        this->_string = value;
    }

/*!
 \brief Get the type of the expression: STRING
 \return int
 \sa          print
*/
    int getType();

/*!
 \brief Print the expression
 \return void

```

```

        \sa          evaluate()
    */
    void print();

    /*!
        \brief      Evaluate the expression
        \return     double
        \sa          print
    */
    std::string evaluateString();
};

```

Listing 90: Método *void lp::StringNode::getType()*

```

int lp::StringNode::getType()
{
    return STRING;
}

```

Listing 91: Método *void lp::StringNode::print()*

```

void lp::StringNode::print()
{
    std::cout << "StringNode: " << this->_string << std::endl;
}

```

Listing 92: Método *void lp::StringNode::evaluate()*

```

std::string lp::StringNode::evaluateString()
{
    return this->_string;
}

```

#### 6.2.4 StringOperatorNode

Se han implementado las siguientes funcionalidades:

- La propia clase.
- Métodos públicos: *void lp::StringOperatorNode::getType()*

Listing 93: Clase *StringOperatorNode : public OperatorNode*

```

/*!
    \class      StringOperatorNode
    \brief      Definition of attributes and methods of StringOperatorNode class
    \note      StringOperatorNode Class publicly inherits from OperatorNode class
    \warning   Abstract class, because it does not redefine the print method of
                ExpNode
*/
    class StringOperatorNode : public OperatorNode

```

```

{
public:
    /*!
        \brief Constructor of StringOperatorNode uses OperatorNode's
               constructor as members initializer
        \param L: pointer to ExpNode
        \param R: pointer to ExpNode
        \post A new StringOperatorNode is created with the parameters
    */
    StringOperatorNode(ExpNode *L, ExpNode *R) : OperatorNode(L, R)
    {
        //      Empty
    }

    /*!
        \brief Get the type of the children expressions
        \return int
        \sa      print()
    */
    int getType();
};

```

Listing 94: Método *void lp::StringOperatorNode::getType()*

```

int lp::StringOperatorNode::getType()
{
    int result = 0;

    if ((this->_left->getType() == STRING) and (this->_right->getType() ==
        STRING))
        result = STRING;
    else
        warning("Runtime error: incompatible types for", "String Operator"
            );

    return result;
}

```

## 7 Funciones auxiliares

### 7.1 Descripción

Se han implementado las siguientes funcionalidades de forma ajena a la lógica del intérprete:

Listing 95: Control del modo de ejecución.

```
switch (argc)
{
    case 1:
        // Execution through interactive mode.
        interactiveMode = true;
        break;

    case 2:
        // Execution through file mode.
        std::string file = argv[1];

        if ((file[file.size() - 2] != '.') or (file[file.size() - 1] != 'e'))
        {
            std::cerr << "[ERROR] The extension of the file \'' << file << '\' is
                not \'.e\'" << std::endl;

            exit(-1);
        }

        else if (file.size() < 3)
        {
            std::cerr << "[ERROR] The name \'' << file << '\' of the file is
                invalid" << std::endl;

            exit(-1);
        }

        else if (access(argv[1], F_OK) != 0)
        {
            std::cerr << "[ERROR] The file \'' << file << '\' does not exist" <<
                std::endl;

            exit(-1);
        }

        yyin = fopen(argv[1], "r");
        interactiveMode = false;
}
```

## 8 Modo de obtención del intérprete

El proyecto se ha dividido en los siguientes directorios y ficheros:

- **Grupo11:** directorio raíz.

- **/ast:** se encuentra el código necesario para generar el árbol de sintaxis abstracta:
  - \* **ast.cpp:** contiene el desarrollo de los métodos de las clases relacionadas con el AST.
  - \* **ast.hpp:** contiene las definiciones de las clases y métodos relacionados con el AST.
  - \* **makefile:** fichero makefile para la compilación del directorio.
- **/error:** se encuentra el código necesario para el control de errores de la gramática del intérprete:
  - \* **error.cpp:** contiene el cuerpo de las funciones relacionadas con el control de errores.
  - \* **error.hpp:** contiene las definiciones de las funciones relacionados con el control de errores.
  - \* **makefile:** fichero makefile para la compilación del directorio.
- **/examples:** se encuentran los distintos ejemplos usados para la verificación del correcto funcionamiento del intérprete.
  - \* **conversion.e:** se verifica que el intérprete sea capaz de cambiar el tipo de dato de una variable.
  - \* **menu.e:** se proporcionan dos algoritmos distintos, seleccionables a partir de un menú interactivo: resolución del factorial de un número y la resolución del máximo común divisor mediante el algoritmo de Euclides.
  - \* **test\_#\_y\_mayusculas.e:** se verifica que las variables no puedan tomar el nombre de una palabra reservada. A su vez, se comprueba que dichas variables no sean sensibles al uso de mayúsculas-minúsculas.
  - \* **test\_booleano.e:** se verifica el funcionamiento de valores *booleanos*.
  - \* **test\_cadenas.e:** se verifica el funcionamiento de las cadenas.
  - \* **test\_casos.e:** se verifica el funcionamiento de la estructura condicional CASOS.
  - \* **test\_error.txt:** se verifica que el intérprete únicamente lea ficheros de ejemplo con extensión “.e”.
  - \* **test\_mientras.e:** se verifica el funcionamiento de la estructura de bucle MIEN-TRAS.
  - \* **test\_numeros.e:** se verifican todas las opciones de lectura de valores de tipo numérico.
  - \* **test\_operadores\_aritmeticos.e:** se verifica el funcionamiento de los operadores aritméticos, especificados con anterioridad en la sección 2.1.

- \* ***test\_operadores\_relacionales.e***: se verifica el funcionamiento de los operadores relacionales, especificados con anterioridad en la sección 2.1.
  - \* ***test\_palabras\_reservadas.e***: se verifica el funcionamiento de las palabras reservadas, especificadas con anterioridad en la sección 2.1.
  - \* ***test\_para.e***: se verifica el funcionamiento de la estructura de bucle PARA.
  - \* ***test\_repetir.e***: se verifica el funcionamiento de la estructura de bucle REPETIR.
  - \* ***test\_si.e***: se verifica el funcionamiento de la estructura condicional SI.
- **/includes**:
- \* ***macros.hpp***: contiene las definiciones de las distintas macros.
- **/parser**: se encuentran los ficheros referentes al análisis tanto sintáctico como léxico del intérprete.
- \* ***interpreter.l***: fichero fuente en código *Flex* con el análisis léxico del intérprete.
  - \* ***interpreter.y***: fichero fuente en código *Bison* con el análisis sintáctico del intérprete.
  - \* ***makefile***: fichero makefile para la compilación del directorio.
- **/table**: se encuentran los ficheros con las distintas clases usadas por la tabla de símbolos. Debido a la extensión de este directorio, se procederá a señalar el objetivo de cada par de ficheros “.cpp “hpp”:
- \* ***builtin***: clase padre que define las funciones virtuales que posteriormente usarán las clases hijas para implementar funciones con 0, 1 o 2 argumentos.
  - \* ***builtinParameter0***: construye funciones con 0 parámetros. Hereda de *builtin*.
  - \* ***builtinParameter1***: construye funciones con 1 parámetro. Hereda de *builtin*.
  - \* ***builtinParameter2***: construye funciones con 2 parámetros. Hereda de *builtin*.
  - \* ***constant***: creación de constantes.
  - \* ***init***: inicializa elementos de la tabla de símbolos.
  - \* ***keyword***: creación de palabras clave.
  - \* ***logicalConstant***: uso de constantes lógicas. Hereda de *constant*.
  - \* ***logicalVariable***: uso de variables lógicas. Hereda de *variable*.
  - \* ***makefile***: fichero makefile para la compilación del directorio.
  - \* ***mathFunction***: funciones matemáticas.
  - \* ***numericConstant***: uso de constantes numéricas. Hereda de *constant*.
  - \* ***numericVariable***: uso de variables numéricas. Hereda de *variable*.
  - \* ***symbol***: clase heredera de *symbolInterface*, padre de *builtin*, *constant*, *keyword* y *variable*, que permite su incorporación a la tabla de símbolos.
  - \* ***symbolInterface***: clase abstracta. *symbolInterface* hereda de ella.
  - \* ***table***: crea la tabla de símbolos. Hereda de *tableInterface*.
  - \* ***tableInterface***: clase abstracta. *table* hereda de ella.
  - \* ***variable***: creación de variables.

- **Doxyfile:** fichero generador de la documentación en *Doxygen*.
- **interpreter.cpp:** fichero fuente con el programa principal.
- **makefile:** fichero makefile para la compilación del intérprete.



## 9 Modo de ejecución

En lo que al modo de ejecución se refiere, el usuario podrá optar por dos opciones diferenciadas: ejecución interactiva o ejecución a partir de un fichero con extensión “.e”.

### 9.1 Interactiva

El modo de ejecución interactiva permite al usuario hacer uso del intérprete desde la consola, de tal manera que, de manera constante, se le puedan enviar órdenes en tiempo de ejecución. De esta manera se consigue que, con una única ejecución podamos ejecutar varios pseudocódigos, tal y como se puede apreciar en la siguiente figura:



```
elon-musk-tweet@elonmusktweet: ~/Escritorio/Engineering/Assignment_LP/Gr...
elon-musk-tweet@elonmusktweet:~/Escritorio/Engineering/Assignment_LP/Grupo11$ ./interpreter.exe

dato := 3;
repetir ++dato; escribir(dato); hasta(dato = 10);
4
5
6
7
8
9
10

dato := 'Procesadores de Lenguajes';
escribir_cadena(dato);
Procesadores de Lenguajes
```

Figura 5: Ejecución interactiva del intérprete.

En caso de que el usuario quiera terminar con la ejecución del intérprete, puede hacer uso en todo momento de la combinación de teclas *Ctrl + D* o, de forma alternativa, si se desea matar el proceso *Ctrl + C*.


## 9.2 A partir de un fichero

Por otro lado, en caso de que el usuario quiera importar un ejemplo particular, siempre se puede hacer uso del paso de argumentos por línea de comandos, de tal manera que se referencie el fichero con extensión “.e” deseado. Para ello, bastaría hacerlo de la siguiente manera:

Listing 96: **Ejecución del intérprete a partir de un fichero.**

```
./interpreter.cpp <fichero.e>
```

El resultado esperado con el ejemplo *test\_repetir.e* sería el siguiente:



```
elon-musk-tweet@elonmusktweet: ~/Escritorio/Engineering/Assignment_LP/Gr...
elon-musk-tweet@elonmusktweet:~/Escritorio/Engineering/Assignment_LP/Grupo11$ ./interpreter.exe
examples/
conversion.e          test_operadores_aritmeticos.e
menu.e               test_operadores_relacionales.e
test_booleano.e      test_palabras_reservadas.e
test_cadenas.e       test_para.e
test_casos.e         test_repetir.e
test_error.txt       test_si.e
test_mientras.e      test_#_y_mayusculas.e
test_numeros.e
elon-musk-tweet@elonmusktweet:~/Escritorio/Engineering/Assignment_LP/Grupo11$ ./interpreter.exe
examples/test_repetir.e
Introduzca un número: 1
2
3
4
5
6
7
8
9
10
elon-musk-tweet@elonmusktweet:~/Escritorio/Engineering/Assignment_LP/Grupo11$
```

Figura 6: Ejecución del intérprete desde fichero.

### 9.2.1 Control de errores

Para prevenir que el usuario introduzca como argumento ficheros no válidos o ficheros inexistentes, se ha realizado el siguiente control de errores: véase sección 7.1 → *Control del modo de ejecución*.

## 10 Ejemplos

A la hora de comprobar si nuestro intérprete cumplía con los requerimientos y verificar si satisfacía las funcionalidades, se han usado algunos ejemplos, algunos preestablecidos y otros creados por nosotros mismos<sup>4</sup>:

### 10.1 menu.e

Fichero de ejemplo del profesor con un pequeño programa que hace factoriales y el máximo común divisor de dos números.

Listing 97: Ejemplos - *menu.e*

```
<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en espa ol: ipe.exe
>>

@ Bienvenida

#borrar;

#lugar(10,10);

escribir_cadena('Introduce tu nombre --> ');

leer_cadena(nombre);

#borrar;
#lugar(10,10);

escribir_cadena(' Bienvenido/a << ');

escribir_cadena(nombre);

escribir_cadena(' >> al intérprete de pseudocódigo en espa ol:\'ipe.exe\'.');
```

---

<sup>4</sup>Puede encontrar los siguientes ejemplos en la carpeta *Grupo11/examples*.

```
#lugar(40,10);
escribir_cadena('Pulsa una tecla para continuar');
leer_cadena( pausa);

repetir

@ Opciones disponibles

#borrar;

#lugar(10,10);
escribir_cadena(' Factorial de un número --> 1 ');

#lugar(11,10);
escribir_cadena(' Máximo común divisor ----> 2 ');

#lugar(12,10);
escribir_cadena(' Finalizar -----> 0 ');

#lugar(15,10);
escribir_cadena(' Elige una opcion ');

leer(opcion);

#borrar;

@ Fin del programa
si (opcion = 0)
    entonces
        #lugar(10,10);
        escribir_cadena(nombre);
        escribir_cadena(': gracias por usar el intérprete ipe.exe ');

@ Factorial de un número
si_no
    si (opcion = 1)
        entonces
            #lugar(10,10);
            escribir_cadena(' Factorial de un numero ');

            #lugar(11,10);
            escribir_cadena(' Introduce un numero entero ');
            leer(N);

            factorial := 1;

            para i desde 2 hasta N paso 1 hacer
                factorial := factorial * i;
            fin_para;

            @ Resultado
            #lugar(15,10);
```

```
        escribir_cadena(' El factorial de ');
        escribir(N);
        escribir_cadena(' es ');
        escribir(factorial);

@ Máximo común divisor
si_no
    si (opcion = 2)
        entonces
            #lugar(10,10);
            escribir_cadena(' Máximo común divisor de dos nú
                meros ');

            #lugar(11,10);
            escribir_cadena(' Algoritmo de Euclides ');

            #lugar(12,10);
            escribir_cadena(' Escribe el primer número ');
            leer(a);

            #lugar(13,10);
            escribir_cadena(' Escribe el segundo número ');
            leer(b);

        @ Se ordenan los números
        si (a < b)
            entonces
                auxiliar := a;
                a := b;
                b := auxiliar;
        fin_si;

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

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

        mientras (resto <> 0) hacer
            a := b;
            b := resto;
            resto := a #mod b;
        fin_mientras;

        @ Se muestra el resultado
        #lugar(15,10);
        escribir_cadena(' Máximo común divisor
            de ');
        escribir(A1);
        escribir_cadena(' y ');
        escribir(B1);
        escribir_cadena(' es ---> ');
```

```

                                escribir(b);

                                @ Resto de opciones
                                si_no
                                    #lugar(15,10);
                                    escribir_cadena(' Opcion incorrecta ');

                                fin_si;

                                fin_si;

                                fin_si;

                                #lugar(40,10);
                                escribir_cadena('\n Pulse una tecla para continuar --> ');
                                leer_cadena(pausa);

                                hasta (opcion = 0);

                                @ Despedida final

                                #borrar;
                                #lugar(10,10);
                                escribir_cadena('El programa ha concluido');
    
```

## 10.2 conversion.e

Fichero de ejemplo del profesor para comprobar al cambio dinámico de tipos.

Listing 98: Ejemplos - *conversion.e*

```

<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

#borrar;

#lugar(3,10);
escribir_cadena('Ejemplo de cambio del tipo de valor \n');
    
```

```

escribir_cadena('Introduce un número --> ');
leer(dato);

escribir_cadena('El número introducido es -> ');
escribir(dato);

escribir_cadena('Introduce una cadena de caracteres --> ');
leer_cadena(dato);

escribir_cadena('La cadena introducida es -> ');
escribir_cadena(dato);

#lugar(20,10);
escribir_cadena(' Fin del ejemplo de cambio del tipo de valor \n');
```

### 10.3 test\_#\_y\_mayusculas.e

Ejemplo del uso de ciertas palabras reservadas y de cómo las variables son indiferentes a las mayúsculas y minúsculas.

#### Listing 99: Ejemplos - *test\_#\_y\_mayusculas.e*

```

<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@Ejemplo del uso de las palabras reservadas y las variables indiferentes a mayúsculas y minúsculas

Dato:=10 #mod 5;
escribir(dAto);

daTo:=10 #div 5;
escribir(dat0);

si ((Dato < 100) #y (daT0 > 1))
    entonces
```

```

                escribir_cadena('Primero \n');
si_no
    si ((DATO < 100) #y #no (DaTo > 1))
        entonces
            escribir_cadena('Segundo \n');
        si_no
            si ((dAtO < 100) #o (dATO > 1))
                entonces
                    escribir_cadena('Tercero \n');
            fin_si;
        fin_si;
fin_si;

#borrar;
#lugar(3,5);
    
```

## 10.4 test\_booleano.e

Ejemplo del uso de los booleanos *verdadero* y *falso*.

### Listing 100: Ejemplos - *test\_booleano.e*

```

<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@Ejemplo del uso de los booleanos

dato:=falso;

si (dato = verdadero)
    entonces
        escribir_cadena('Es verdadero\n');
    si_no
        si (dato = falso)
            entonces
                escribir_cadena('Es falso\n');
            si_no
    
```



```

                                escribir_cadena('La variable no es un booleano\n
                                ');
                                fin_si;
fin_si;

```

## 10.5 test\_cadenas.e

Ejemplo del uso de cadenas y las funciones *escribir\_cadena* y *leer\_cadena*. También se muestra el uso de la concatenación.

Listing 101: Ejemplos - *test\_cadenas.e*

```

<<
  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: 2020 - 2021

  Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@Ejemplo del uso de cadenas

escribir_cadena('Escribe un número --> ');
leer(num);
escribir_cadena('Escribe una cadena --> ');
leer_cadena(cad);

escribir_cadena('Número: ');
escribir(num);
escribir_cadena('Cadena: ');
escribir_cadena(cad);

escribir_cadena('\n Ejemplo de cadena consalto de línea \n y tabulador \t \n');
escribir_cadena('Ejemplo de cadena con \' comillas\' simples \n');

dato:= 'hola' || ' adios';
escribir_cadena(dato);
escribir_cadena('\n');

```

## 10.6 test\_casos.e

Fichero ejemplo del uso del bucle *casos*.

Listing 102: Ejemplos - *test\_casos.e*

```
<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@Ejemplo del uso del bucle CASOS

escribir_cadena('Introduzca un número:');
leer(dato);

casos (dato)
    valor 1:
        escribir_cadena('Número igual a 1\n');
    valor 2:
        escribir_cadena('Número igual a 2\n');
    defecto:
        escribir_cadena('Número distinto de 1 y 2\n');
fin_casos;

escribir_cadena('Introduzca \'hola\' , \'adios\' u otra cadena:');
leer_cadena(dato);

casos (dato)
    valor 'hola':
        escribir_cadena('Cadena igual a \'hola\'\n');
    valor 'adios':
        escribir_cadena('Cadena igual a \'adios\'\n');
    defecto:
        escribir_cadena('Cadena distinta de \'hola\' y \'adios\'\n');
fin_casos;

dato:=falso;

casos (dato)
    valor verdadero:
        escribir_cadena('El booleano es verdadero\n');
    valor falso:
        escribir_cadena('El booleano es falso\n');
    defecto:
```

```
        escribir_cadena('Error\n');
fin_casos;
```

## 10.7 test\_error.txt

Ejemplo de cómo un archivo que no tiene extensión ".e", no puede ser ejecutado por el intérprete.

Listing 103: Ejemplos - *test\_error.txt*

```
<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@Ejemplo de como un archivo, que no tiene extensión ".e", no puede ser ejecutado
por el intérprete.
```

## 10.8 test\_numeros.e

Fichero de ejemplo sobre el correcto funcionamiento de los numeros enteros, decimales y en notación científica.

Listing 104: Ejemplos - *test\_numeros.e*

```
<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe  
>>

@ Descripción: Ejemplo de números disponibles

```
dato:=1;
escribir(dato);
dato:=1.2;
escribir(dato);
dato:=1.2E+3;
escribir(dato);
dato:=1.2E-3;
escribir(dato);
```

## 10.9 test\_operadores\_aritmeticos.e

Ejemplo de uso de los operadores aritméticos, así como los operadores extra  $++$ ,  $-$ ,  $+=$  y  $-=$ .

### Listing 105: Ejemplos - *test\_operadores\_aritmeticos.e*

<<

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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe  
>>

@ Descripción: Ejemplo de operadores aritméticos

```
dato:= +4;
escribir(dato);
dato:= -4;
escribir(dato);
dato:= 3 + 4;
escribir(dato);
dato:= 3 - 4;
escribir(dato);
dato:= 3 * 4;
```

```
escribir(dato);
dato:= 3 / 4;
escribir(dato);
dato:= 3 ** 4;
escribir(dato);
```

@ Descripción: Ejemplo del uso de los operadores extra

```
++dato;
escribir(dato);
--dato;
escribir(dato);
dato+= 3;
escribir(dato);
dato-= 3;
escribir(dato);
```

## 10.10 test\_operadores\_relacionales.e

Ejemplo de cómo usar los operadores relacionales del intérprete.

Listing 106: Ejemplos - *test\_operadores\_relacionales.e*

```
<<
  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: 2020 - 2021

  Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@ Descripción: Ejemplo de operadores relacionales

@Operando sobre cadenas

escribir_cadena('Introduce la primera cadena: \n');
leer_cadena(n1);

escribir_cadena('Introduce la segunda cadena: \n');
leer_cadena(n2);

escribir_cadena('Cadena n1: ');
```

```

escribir_cadena(n1);
escribir_cadena('\n');

escribir_cadena('Cadena n2: ');
escribir_cadena(n2);
escribir_cadena('\n');

si(n1 < n2) entonces
escribir_cadena('Success: n1 < n2 \n');
fin_si;

si(n1 > n2) entonces
escribir_cadena('Success: n1 > n2 \n');
fin_si;

si(n1 <= n2) entonces
escribir_cadena('Success: n1 <= n2 \n');
fin_si;

si(n1 >= n2) entonces
escribir_cadena('Success: n1 >= n2 \n');
fin_si;

si(n1 = n2) entonces
escribir_cadena('Success: n1 = n2 \n');
fin_si;

si(n1 <> n2) entonces
escribir_cadena('Success: n1 <> n2 \n');
fin_si;

```

### 10.11 test\_palabras\_reservadas.e

Fichero de ejemplo de como las palabras reservadas no son identificadores y por lo tanto no se pueden asignar a variables.

Listing 107: Ejemplos - *test\_palabras\_reservadas.e*

```

<<
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: 2020 - 2021

```

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe  
>>

@TEST: LAS PALABRAS RESERVADAS NO SON IDENTIFICADORES

```
dato := #mod;  
leer_cadena(dato);
```

```
dato := #div;  
leer_cadena(dato);
```

```
dato := #o;  
leer_cadena(dato);
```

```
dato := #y;  
leer_cadena(dato);
```

```
dato := #no;  
leer_cadena(dato);
```

```
dato := verdadero;  
leer_cadena(dato);
```

```
dato := falso;  
leer_cadena(dato);
```

```
dato := leer;  
leer_cadena(dato);
```

```
dato := leer_cadena;  
leer_cadena(dato);
```

```
dato := escribir;  
leer_cadena(dato);
```

```
dato := escribir_cadena;  
leer_cadena(dato);
```

```
dato := si;  
leer_cadena(dato);
```

```
dato := entonces;  
leer_cadena(dato);
```

```
dato := si_no;  
leer_cadena(dato);
```

```
dato := fin_si;  
leer_cadena(dato);
```

```
dato := mientras;  
leer_cadena(dato);
```

```
dato := hacer;
leer_cadena(dato);

dato := fin_mientras;
leer_cadena(dato);

dato := repetir;
leer_cadena(dato);

dato := hasta;
leer_cadena(dato);

dato := para;
leer_cadena(dato);

dato := desde;
leer_cadena(dato);

dato := paso;
leer_cadena(dato);

dato := casos;
leer_cadena(dato);

dato := valor;
leer_cadena(dato);

dato := fin_casos;
leer_cadena(dato);

dato := defecto;
leer_cadena(dato);

dato := #borrar;
leer_cadena(dato);

dato := #lugar;
leer_cadena(dato);

dato := casos;
leer_cadena(dato);

dato := valor;
leer_cadena(dato);

dato := fin_casos;
leer_cadena(dato);

dato := defecto;
leer_cadena(dato);
```



## 10.12 test\_para.e

Fichero ejemplo del uso del bucle *para*.

### Listing 108: Ejemplos - *test\_para.e*

```
<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe
>>

@ Descripción: Ejemplo del bucle PARA

dato1:=0;
escribir_cadena('Introduzca un número: ');
leer(dato);
escribir dato;
k:=8;

para k
    desde 1
    hasta 10
    paso 1
    hacer
        dato += 1;
        dato1 += 2;
fin_para;

escribir dato;
escribir dato1;
```

## 10.13 test\_repetir.e

Fichero ejemplo del uso del bucle *repetir*.

### Listing 109: Ejemplos - *test\_repetir.e*

```
<<
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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe  
>>

@ Descripción: Ejemplo del uso del bucle REPETIR

```
escribir_cadena('Introduzca un número: ');  
leer(dato);
```

```
repetir  
    dato := dato + 1;  
    escribir(dato);  
hasta(dato = 10);
```

## 10.14 test\_si.e

Fichero ejemplo del uso del condicional *si*.

### Listing 110: Ejemplos - *test\_si.e*

<<

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: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español: ipe.exe  
>>

@ Descripción: Ejemplo del uso del condicional SI

```
dato:=6;
```

```
si (dato > 5)
    entonces
        dato+=10;
        escribir_cadena('Estamos dentro del entonces \n');
        escribir(dato);
    si_no
        dato-=10;
        escribir_cadena('Estamos dentro del si_no \n');
        escribir(dato);
fin_si;
```

## 11 Conclusiones

### 11.1 Conclusiones generales

Tras terminar la práctica hace poco, tenemos la impresión de que nuestro trabajo ha sido el esperado teniendo en cuenta la cantidad de tiempo empleado en ella. Aunque el intérprete desde luego no es perfecto, sabemos que tiene ciertos problemas de control de errores y cierto código que se podría refactorizar para mejorar su organización y calidad. Por otro lado, hemos desarrollado bastantes de las funcionalidades opcionales, como la de la función *switch* o los operadores `++`, `--`, `+=` y `-=`, que nos resultaron muy problemáticos en su momento. Sentimos que nos ha quedado relativamente claro el contenido de las prácticas de esta asignatura y tenemos una visión global del procesamiento de lenguajes mayor de con la que empezamos, por lo que nos sentimos satisfechos con nuestro trabajo.

### 11.2 Puntos fuertes y puntos débiles del intérprete

En primer lugar, creemos que nuestro intérprete funciona correctamente, es decir, cumple con la mayoría de los requerimientos solicitados en el enunciado de la práctica. Todos y aquellos errores que fueron encontrados durante la realización fueron corregidos de tal manera que, aparentemente, no queda ninguno, aunque sí que pueden existir algunos no detectables por el compilador.

Es por ello que consideramos que nuestro intérprete es a simple vista robusto, aunque se podría haber aumentado el control de errores. Además, tal y como hemos comentado con anterioridad, al desarrollar implementaciones extra, creemos proporciona libertad y diversidad de opciones.

### 11.3 Otras consideraciones

Finalmente, queríamos remarcar el gran esfuerzo y la gran cantidad de tiempo empleada en el proyecto. Creemos que, en gran medida, el tiempo que hay que dedicarle es excesivo, debido a la entrega no sólo del código, sino también del presente informe. Entendemos que hay que dejar constancia de todo el trabajo realizado, pero quizás se pueda plantear de una manera en la que no consuma tanto tiempo al alumnado, sobretodo a escasas semanas de los exámenes finales.

## Referencias

- [1] Moodle Universidad de Córdoba / Grado en Ingeniería Informática / Asignatura Procesadores de Lenguajes, apartado de prácticas de la asignatura, Año 2020-2021 - Nicolás Luis Fernández García.