



UNIVERSIDAD  
DE  
CÓRDOBA

ESCUELA POLITÉCNICA  
SUPERIOR DE CÓRDOBA  
Universidad de Córdoba



# Intérprete de pseudocódigo en español: IPE

PROCESADORES DE LENGUAJE  
INGENIERÍA INFORMÁTICA  
ESPECIALIDAD DE COMPUTACIÓN  
TERCER CURSO, SEGUNDO CUATRIMESTRE  
ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA  
UNIVERSIDAD DE CÓRDOBA  
CURSO ACADÉMICO: 2020 – 2021

Carlos Freire Caballero  
Carlos Ayuso Sánchez

7 de junio de 2021

# Índice de contenido

	Página
<b>1. Introducción</b>	<b>4</b>
1.1. Descripción del trabajo . . . . .	4
1.2. Partes del documento . . . . .	4
<b>2. Lenguaje de pseudocódigo</b>	<b>5</b>
2.1. Componentes léxicos . . . . .	5
2.1.1. Palabras reservadas . . . . .	5
2.1.2. Identificadores . . . . .	6
2.1.3. Número . . . . .	6
2.1.4. Cadena . . . . .	6
2.1.5. Operador asignación . . . . .	7
2.1.6. Operadores numéricos . . . . .	7
2.1.7. Operadores alfanuméricos . . . . .	7
2.1.8. Operadores relacionales . . . . .	7
2.1.9. Operadores lógicos . . . . .	8
2.2. Sentencias . . . . .	8
2.2.1. Asignación . . . . .	8
2.2.2. Lectura . . . . .	8
2.2.3. Escritura . . . . .	9
2.2.4. Sentencia de control . . . . .	9
2.2.5. Comandos especiales . . . . .	10
<b>3. Tabla de símbolos</b>	<b>11</b>
<b>4. Análisis léxico</b>	<b>13</b>
4.1. Componentes léxicos y expresiones regulares . . . . .	13
<b>5. Análisis sintáctico</b>	<b>15</b>

5.1. Símbolos terminales . . . . .	15
5.2. Símbolos no terminales . . . . .	16
5.3. Reglas de producción . . . . .	16
5.4. Acciones semánticas . . . . .	19
<b>6. Árbol sintáctico abstracto</b>	<b>29</b>
<b>7. Modo de obtención del intérprete</b>	<b>32</b>
<b>8. Modo de ejecución</b>	<b>35</b>
8.1. Ejecución interactiva . . . . .	35
8.2. Ejecución a partir de un fichero . . . . .	36
<b>9. Ejemplos</b>	<b>37</b>
9.1. conversion.e . . . . .	37
9.2. ecuacion.e . . . . .	38
9.3. ecuacion-error.e . . . . .	39
9.4. entrada.txt . . . . .	41
9.5. factorial.e . . . . .	41
9.6. factorial-error.e . . . . .	41
9.7. menu.e . . . . .	42
9.8. menuCasos.e . . . . .	45
9.9. op-aritmeticas.e . . . . .	45
9.10. op-relacionales.e . . . . .	47

9.11. op-relacionales-error.e . . . . .	47
9.12. primo.e . . . . .	48
9.13. primo-error.e . . . . .	48
9.14. test2.txt . . . . .	49
<b>10. Conclusiones</b>	<b>51</b>
10.1. Reflexión general . . . . .	51
10.2. Puntos fuertes y puntos débiles del intérprete . . . . .	51
10.2.1. Puntos fuertes . . . . .	51
10.2.2. Puntos débiles . . . . .	51

# Lista de figuras

3.1. Estructura de las clases SymbolInterface y TableInterface . . . . .	11
6.1. Estructura de la clase Statement . . . . .	29
6.2. Estructura de la clase ExpNode . . . . .	31
8.1. Ejecución interactiva del intérprete . . . . .	35
8.2. Ejecución a partir del fichero factorial.e del intérprete . . . . .	36

# Introducción

## 1.1. Descripción del trabajo

Este trabajo consiste en la creación de un intérprete de pseudocódigo en español, haciendo uso de los lenguajes *Flex* [1] y *Bison* [2]. Dentro del trabajo aplicaremos los conocimientos adquiridos en la asignatura de *procesadores de lenguajes* [3] sobre el análisis léxico, semántico y sintáctico de los intérpretes.

## 1.2. Partes del documento

- **Lenguaje de pseudocódigo:** donde encontraremos tanto los componentes léxicos como las sentencias usadas.
- **Tabla de símbolos:** muestra y almacena la información relacionada con el programa como variables, funciones, constantes, etc.
- **Análisis léxico:** Consiste en la creación de los tokens que reconocerá nuestro intérprete.
- **Análisis sintáctico:** Se reciben los tokens generados con los componentes léxicos y se comprueba si cumplen las reglas sintácticas de nuestro lenguaje.
- **AST:** El árbol sintáctico abstracto genera la estructura de las clases que utilizamos y que se van almacenando en nuestro intérprete.
- **Funciones auxiliares:** Nuevas funciones que no se habían definido y que generan una mejora al trabajo.
- **Modos de ejecución:** se explican las distintas formas de ejecución del intérprete.
- **Ejemplos:** Recopilación de ejemplos creados por el profesor y diseñados por nosotros mismos para realizar distintas pruebas sobre nuestro intérprete.
- **Conclusiones:** Reflexión sobre el trabajo y sobre los aspectos positivos o negativos que podríamos destacar.

# Lenguaje de pseudocódigo

## 2.1. Componentes léxicos

### 2.1.1. Palabras reservadas

Estas palabras están predefinidas en nuestra tabla de símbolos, el resto de componentes léxicos están definidos en el intérprete. Dentro de la tabla de símbolos también definiremos las constantes numéricas y las funciones.

Para las constantes crearemos una estructura.

---

```
//init.hpp
static struct {
    std::string name ;
    bool value;
} logicalConstant[] = {
    {"verdadero", true},
    {"falso", false},
    {"", 0}
};
```

---

Para las sentencias crearemos otra estructura.

---

```
//init.hpp
static struct {
    std::string name ;
    int token;
} keyword[] = {
    {"PIDE_TECLA", ASK_FOR_KEY},
    {"ESCRIBIR", PRINT},
    {"ESCRIBIR_CADENA", PRINT},
    {"LEER", READ},
    {"LEER_CADENA", READ_STRING},
    {"SI", IF},
    {"ENTONCES", THEN},
    {"FIN_SI", END_IF},
    {"SI_NO", ELSE},
    {"MIENTRAS", WHILE},
    {"HACER", DO},
    {"FIN_MIENTRAS", END_WHILE},
    {"REPETIR", DO_WHILE},
    {"HASTA", UNTIL},
    {"PARA", FOR},
    {"DESDE", FROM},
    {"PASO", STEP},
```

```

{"FIN_PARA", END_FOR},
{"CASOS", SWITCH},
{"VALOR", VALUE},
{"DEFECTO", DEFAULT},
{"FIN_CASOS", END_SWITCH},
{"#BORRAR", CLEAR},
{"#LUGAR", PLACECURSOR},
{"", 0}
};

```

---

### 2.1.2. Identificadores

Este componente léxico se identificará gracias a expresiones regulares, Estarán compuestos por una serie de letras, dígitos y el subrayado. Debe comenzar por una letra y no podrá acabar en un símbolo de subrayado, ni tener dos subrayados seguidos.

---

```

//interpreter.l
DIGIT [0-9]

LETTER [a-zA-Z]

NUMBER {DIGIT}+(\.{DIGIT}+)?(e[+\-]?{DIGIT}+)?

IDENTIFIER {LETTER}(_?({LETTER}|{DIGIT})+)*

```

---

### 2.1.3. Número

Este componente léxico se identificara gracias a expresiones regulares, se reconocerá números enteros, reales de punto fijo y reales con notación científica. Todos son tratados como números.

---

```

//interpreter.l
NUMBER {DIGIT}+(\.{DIGIT}+)?(e[+\-]?{DIGIT}+)?

{NUMBER} {
    /* Conversion of type and sending of the numerical value to the parser */
    yylval.number = atof(yytext);

    return NUMBER;
}

```

---

Dentro del código lo único que se hará es convertir de tipo carácter a tipo numérico.

### 2.1.4. Cadena

Este componente léxico se identificara gracias a expresiones regulares, se reconocerá por una serie de caracteres delimitados por comillas simples, donde se podrá incluir comillas simples.

---

```

//interpreter.l
{STRING} {
    std::string yytextString(yytext);
    int strSize = yytextString.size();
    std::string stringWithoutQuotes = yytextString.substr(1, strSize-2);

    yylval.identifier = strdup(stringWithoutQuotes.c_str());
}

```

---

```
    return STRING;
}
```

---

Las comillas exteriores no se almacenarán como parte de la cadena.

### 2.1.5. Operador asignación

El operador asignación `:=` nos permite asignar un valor a un identificador. Representamos el componente léxico dentro del intérprete.

---

```
//interpreter.l
":=" { return ASSIGNMENT; }
```

---

### 2.1.6. Operadores numéricos

Los operadores numéricos solo afectaran con variables numéricas, no se podrá usar en variables aritméticas o intentar operar dos variables de distinto tipo.

- suma: +
  - Unario: +2
  - Binario: 2+3
- resta: -
  - Unario: -2
  - Binario: 2-3
- producto: \*
- división: /
- división entera: #div
- módulo: #mod
- potencia: \*\*

### 2.1.7. Operadores alfanuméricos

El operador concatenación `||` es el equivalente al operador suma pero para variables cadena. Representaremos el componente léxico dentro del intérprete.

---

```
//interpreter.l
"||" { return CONCATENATION; }
```

---

### 2.1.8. Operadores relacionales

Los operadores relacionales funcionaran para variables numéricas y alfanuméricas, de forma independiente.

Para la relación de las variables alfanuméricas serán respecto al orden del abecedario.

- menor que: <
- menor o igual que: <=



- mayor que: >
- mayor o igual que: >=
- igual que: =
- distinto que: <>

### 2.1.9. Operadores lógicos

Los operadores lógicos se utilizan para conectar distintos valores, identificadores o formulas que nos darán como resultado verdadero falso, según si se cumple o no. Representaremos estos componentes léxicos dentro del intérprete.

---

```
//interpreter.l
"#no"    { return NOT; }
"#o"     { return OR; }
"#y"     { return AND; }
```

---

## 2.2. Sentencias

### 2.2.1. Asignación

Para la asignación usaremos el mismo operador para las expresiones numéricas como para las expresiones alfanuméricas.

- **identificador** := expresión numérica
  - Declara 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 identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.
  - Las expresiones numéricas se formarán con números, variables alfanuméricas y operadores concatenación (||).

### 2.2.2. Lectura

Para lectura utilizaremos dos funciones distintas, como no podemos asignarle un tipo de dato al identificador antes tenemos que hacerlo de esta manera.

- **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).

### 2.2.3. Escritura

Para la escritura solo tendremos que usar una única función, inicialmente existían 2 pero es algo redundante ya que si dentro de la expresión de asignación analizamos de que tipo es, podemos escribirlo de una forma o de otra.

- **Escribir** (expresión numérica/alfanumérica)
  - el valor sera escrito por pantalla.
  - Si es una expresión alfanumérica Se debe permitir la interpretación de comandos de saltos de línea y tabuladores que puedan aparecer.

### 2.2.4. Sentencia de control

En este apartado veremos los pseudocódigos de las sentencias que luego analizaremos en el apartado de análisis léxico

- 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**  
**hasta** condición
- Sentencia **casos**"  
    **casos** (expresión)  
        **valor v1:** ...  
        **valor v2:** ...

...  
defecto: ...  
fin\_casos

### 2.2.5. 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.

# Tabla de símbolos

La tabla de símbolos es una estructura de datos en forma de árbol donde se cada uno de los datos generados en nuestro programa serán siendo clasificados y almacenados en ella. Dentro de nuestra tabla podremos encontrar distintos tipos de datos, partiendo de un dato común conocido como el símbolo y cada vez se vuelven mas específicos.

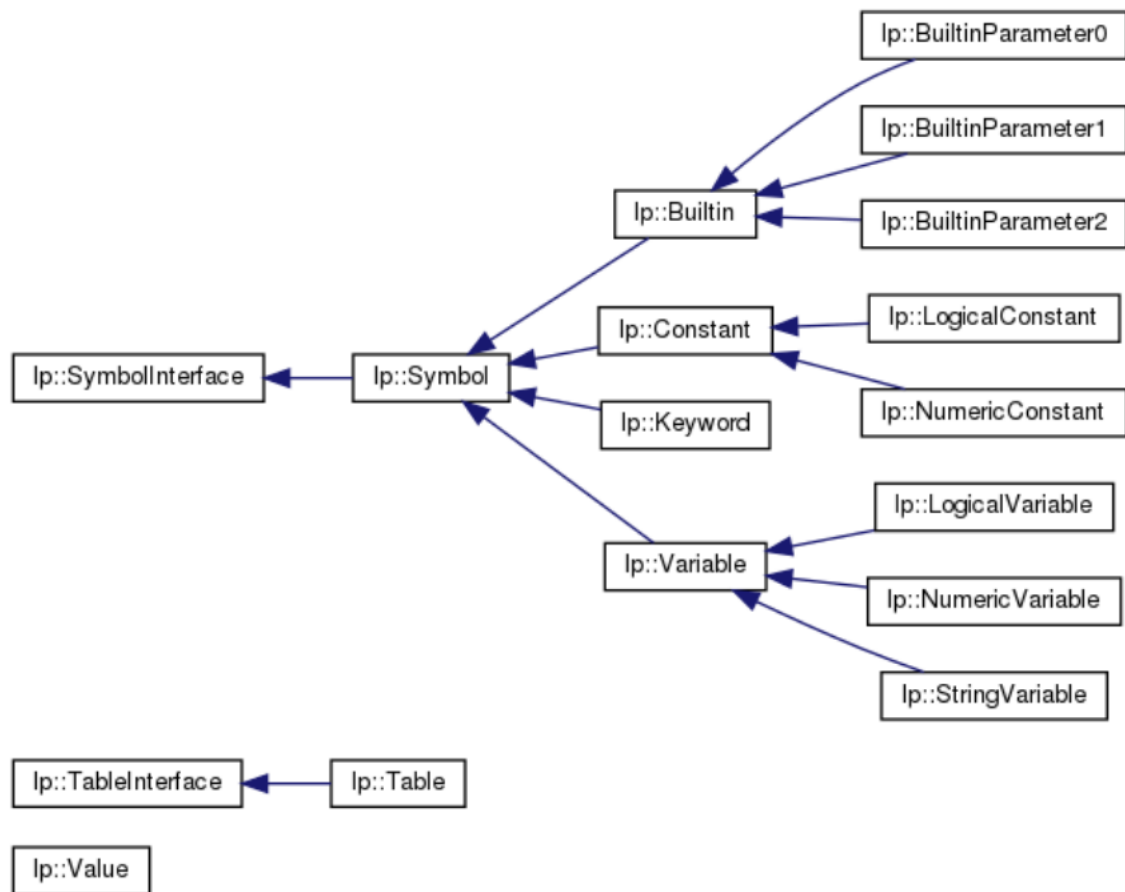


Figura 3.1: Estructura de las clases SymbolInterface y TableInterface

Ahora desglosaremos cada uno de los componentes de la estructura generada:

- **SymbolInterface**

- Symbol
  - Builtin: Clase que define una función predefinida.
    - ◊ builtinParameter0: Clase que define una función predefinida sin parámetros.
    - ◊ builtinParameter1: Clase que define una función predefinida con un parámetro.
    - ◊ builtinParameter2: Clase que define una función predefinida con dos parámetros.
  - Constant: Clase que define las constantes predefinidas.
    - ◊ LogicalConstant: Clase para las constantes lógicas.
    - ◊ NumericConstant: Clase para las constantes numéricas.
  - Keyword: Clase que almacenas las palabras reservadas predefinidas.
  - Variable: Clase que define las variables generadas en la tabla de símbolos.
    - ◊ LogicalVariable: Clase que define las variables como lógicas.
    - ◊ NumericVariable: Clase que define las variables como numéricas.
    - ◊ StringVariable: Clase que define las variables como cadenas.

- Clase **TableInterface**: Clase abstracta con las funciones para acceder a la tabla de símbolos.

- Table: Clase que define la tabla de símbolos.

- Clase **Value**: Clase que representa cada uno de los posibles casos de una sentencia switch.

# Análisis léxico

El primer paso de nuestro intérprete es el de leer carácter a carácter el código fuente con el objetivo de obtener los componentes léxicos para posteriormente enviárselos al analizador sintáctico. Así pues en este apartado se indican todos los componentes léxicos y las expresiones regulares que los denotan.

## 4.1. Componentes léxicos y expresiones regulares

---

<code>";"</code>	<code>{ SEMICOLON }</code>	Delimitador de sentencias.
<code>":"</code>	<code>{ TWO_POINTS }</code>	Se utiliza en los posibles casos de la sentencia <code>switch</code> .
<code>","</code>	<code>{ COMMA }</code>	Se utiliza para separar argumentos en funciones predefinidas
<code>" {DIGIT}+(\.{DIGIT})?(e[+-]?{DIGIT})? "</code>	<code>{ NUMBER }</code>	Representa tanto enteros como reales. DIGIT es un dgito en el intervalo [0,9]
<code>" '\\"([^\"] \\\\"\\\\\\')*\\" '\\" "</code>	<code>{STRING}</code>	Representa a las cadenas que se utilizan en el programa.
<code>"{LETTER}(_?({LETTER} {DIGIT})+)*"</code>	<code>{ IDENTIFIER }</code>	Representa los identificadores del programa.
<code>"-"</code>	<code>{ MINUS }</code>	Operador de resta para nmeros y cadenas
<code>"+"</code>	<code>{ PLUS }</code>	Operador de suma para nmeros y cadenas
<code>"*"</code>	<code>{ MULTIPLICATION }</code>	Operador de multiplicacion para nmeros y cadenas
<code>"/"</code>	<code>{ DIVISION }</code>	Operador de division para nmeros y cadenas
<code>"#div"</code>	<code>{ INTEGER_DIVISION }</code>	Operador de division entera para nmeros y cadenas
<code>"("</code>	<code>{ LPAREN }</code>	Parentesis izquierdo utilizado para agrupar expresiones, en funciones y condiciones
<code>")"</code>	<code>{ RPAREN }</code>	Parentesis derecho utilizado para agrupar expresiones, en funciones y condiciones
<code>"#mod"</code>	<code>{ MODULO }</code>	Operador de mdulo
<code>"**"</code>	<code>{ POWER; }</code>	Operador de potencia
<code>"::="</code>	<code>{ ASSIGNMENT }</code>	Operador de asignacin
<code>"  "</code>	<code>{ CONCATENATION }</code>	Operador de concatenacin

```

"="          { EQUAL } Operador de comparacin
"<>"        { NOT_EQUAL } Operador de distinto a
">="         { GREATER_OR_EQUAL } Operador de mayor o igual
"<="         { LESS_OR_EQUAL } Operador de menor o igual
">"          { GREATER_THAN } Operador de mayor
"<"          { LESS_THAN } Operador de menor

(?i:"#no") { NOT } Operador de negacin
(?i:"#o") { OR } Operador or
(?i:"#y") { AND } Operador and

"{"          { LETFCURLYBRACKET } Se inicia al comienzo de listas de sentencias. En la prctica no
se ha utilizado.
"}"          { RIGHTCURLYBRACKET } Se inicia al final de listas de sentencias. En la prctica no
se ha utilizado.

(?i:#borrar) { CLEAR } Palabra clave para borrar la pantalla de la terminal
(?i:#lugar)   { PLACECURSOR } Palabra clave para posicionar el cursor en coordenadas de la
pantalla de la terminal

```

---

# Análisis sintáctico

## 5.1. Símbolos terminales

Los símbolos terminales son los componentes léxicos definidos en nuestra gramática. Están definidos según el componente léxico o su posición.

- `%token`
- `%left`
- `%right`
- `%nonassoc`

---

```
//interpreter.y
%token PLACECURSOR CLEAR
%token SEMICOLON
%token ASK_FOR_KEY PRINT READ READ_STRING IF THEN END_IF ELSE WHILE DO END_WHILE DO_WHILE
    UNTIL FOR FROM STEP END_FOR
%token LETFCURLYBRACKET RIGHTCURLYBRACKET
%token COMMA
%token <number> NUMBER
%token <identifier> STRING
%token <logic> BOOL
%token <identifier> VARIABLE UNDEFINED CONSTANT BUILTIN

%left OR
%left AND
%left PLUS MINUS CONCATENATION
%left MULTIPLICATION DIVISION INTEGER_DIVISION MODULO
%left LPAREN RPAREN

%right ASSIGNMENT
%right POWER

%nonassoc GREATER_OR_EQUAL LESS_OR_EQUAL GREATER_THAN LESS_THAN EQUAL NOT_EQUAL
%nonassoc UNARY
```

---



## 5.2. Símbolos no terminales

los símbolos terminales estarán comprendidos por su tipos:

- **expNode**
- **parameters**
- **stmts**
- **st**
- **prog**

---

```
//interpreter.y
%type <expNode> exp cond

%type <parameters> listOfExp restOfListOfExp

%type <stmts> stmtlist

%type <switchValues> listOfValues restOfListOfValues

%type <st> stmt asgn ask_for_key print read read_string if while block do_while for switch
           placecursor clear

%type <prog> program
```

---

## 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.  $\text{stmt} \rightarrow \text{ask\_for\_key SEMICOLON}$
8.  $\text{stmt} \rightarrow \text{print SEMICOLON}$
9.  $\text{stmt} \rightarrow \text{read SEMICOLON}$
10.  $\text{stmt} \rightarrow \text{read\_string SEMICOLON}$
11.  $\text{stmt} \rightarrow \text{if SEMICOLON}$
12.  $\text{stmt} \rightarrow \text{while SEMICOLON}$
13.  $\text{stmt} \rightarrow \text{do\_while SEMICOLON}$

14.  $\text{stmt} \rightarrow \text{for SEMICOLON}$
15.  $\text{stmt} \rightarrow \text{switch SEMICOLON}$
16.  $\text{stmt} \rightarrow \text{block SEMICOLON}$
17.  $\text{stmt} \rightarrow \text{placecursor SEMICOLON}$
18.  $\text{stmt} \rightarrow \text{clear SEMICOLON}$
19.  $\text{block} \rightarrow \text{LEFTCURLYBRACKET stmtlist RIGHTCURLYBRACKET}$
20.  $\text{controlSymbol} \rightarrow \epsilon$
21.  $\text{if} \rightarrow \text{IF controlSymbol cond THEN stmtlist ENDIF}$
22.  $\text{if} \rightarrow \text{IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF}$
23.  $\text{while} \rightarrow \text{WHILE controlSymbol cond DO stmtlist END\_WHILE}$
24.  $\text{do\_while} \rightarrow \text{DO\_WHILE controlSymbol stmtlist UNTIL cond}$
25.  $\text{for} \rightarrow \text{FOR controlSymbol VARIABLE FROM exp UNTIL exp STEP exp DO stmtlist END\_FOR}$
26.  $\text{for} \rightarrow \text{FOR controlSymbol VARIABLE FROM exp UNTIL exp DO stmtlist END\_FOR}$
27.  $\text{switch} \rightarrow \text{SWITCH controlSymbol LPAREN exp RPAREN listOfValues DEFAULT TWO\_POINTS stmtlist END\_SWITCH}$
28.  $\text{listOfValues} \rightarrow \text{VALUE NUMBER TWO\_POINTS stmtlist restOfListOfValues}$
29.  $\text{listOfValues} \rightarrow \text{VALUE STRING TWO\_POINTS stmtlist restOfListOfValues}$
30.  $\text{restOfListOfValues} \rightarrow \epsilon$
31.  $\text{restOfListOfValues} \rightarrow \text{VALUE NUMBER TWO\_POINTS stmtlist restOfListOfValues}$
32.  $\text{restOfListOfValues} \rightarrow \text{VALUE STRING TWO\_POINTS stmtlist restOfListOfValues}$
33.  $\text{cond} \rightarrow \text{LPAREN exp RPAREN}$
34.  $\text{asgn} \rightarrow \text{VARIABLE ASSIGNMENT exp}$
35.  $\text{asgn} \rightarrow \text{VARIABLE ASSIGNMENT asgn}$
36.  $\text{asgn} \rightarrow \text{CONSTANT ASSIGNMENT exp}$
37.  $\text{asgn} \rightarrow \text{CONSTANT ASSIGNMENT asgn}$
38.  $\text{ask\_for\_key} \rightarrow \text{ASK\_FOR\_KEY LPAREN exp RPAREN}$
39.  $\text{print} \rightarrow \text{PRINT exp}$
40.  $\text{read} \rightarrow \text{READ LPAREN VARIABLE RPAREN}$
41.  $\text{read} \rightarrow \text{READ LPAREN CONSTANT RPAREN}$
42.  $\text{placecursor} \rightarrow \text{PLACECURSOR LPAREN exp COMMA exp RPAREN}$
43.  $\text{clear} \rightarrow \text{CLEAR}$
44.  $\text{read\_string} \rightarrow \text{READ\_STRING LPAREN VARIABLE RPAREN}$
45.  $\text{read\_string} \rightarrow \text{READ\_STRING LPAREN CONSTANT RPAREN}$

- 46.  $\text{exp} \rightarrow \text{NUMBER}$
- 47.  $\text{exp} \rightarrow \text{STRING}$
- 48.  $\text{exp} \rightarrow \text{exp PLUS exp}$
- 49.  $\text{exp} \rightarrow \text{exp MINUS exp}$
- 50.  $\text{exp} \rightarrow \text{exp MULTIPLICATION exp}$
- 51.  $\text{exp} \rightarrow \text{exp DIVISION exp}$
- 52.  $\text{exp} \rightarrow \text{exp INTEGER\_DIVISION exp}$
- 53.  $\text{exp} \rightarrow \text{exp CONCATENATION exp}$
- 54.  $\text{exp} \rightarrow \text{LPAREN exp RPAREN}$
- 55.  $\text{exp} \rightarrow \text{PLUS exp \%prec UNARY}$
- 56.  $\text{exp} \rightarrow \text{MINUS exp \%prec UNARY}$
- 57.  $\text{exp} \rightarrow \text{exp MODULO exp}$
- 58.  $\text{exp} \rightarrow \text{exp POWER exp}$
- 59.  $\text{exp} \rightarrow \text{VARIABLE}$
- 60.  $\text{exp} \rightarrow \text{CONSTANT}$
- 61.  $\text{exp} \rightarrow \text{BUILTIN LPAREN listOfExp RPAREN}$
- 62.  $\text{exp} \rightarrow \text{exp GREATER\_THAN exp}$
- 63.  $\text{exp} \rightarrow \text{exp GREATER\_OR\_EQUAL exp}$
- 64.  $\text{exp} \rightarrow \text{exp LESS\_THAN exp}$
- 65.  $\text{exp} \rightarrow \text{exp LESS\_OR\_EQUAL exp}$
- 66.  $\text{exp} \rightarrow \text{exp EQUAL exp}$
- 67.  $\text{exp} \rightarrow \text{exp NOT\_EQUAL exp}$
- 68.  $\text{exp} \rightarrow \text{exp AND exp}$
- 69.  $\text{exp} \rightarrow \text{exp OR exp}$
- 70.  $\text{exp} \rightarrow \text{NOT exp}$
- 71.  $\text{listOfExp} \rightarrow \epsilon$
- 72.  $\text{listOfExp} \rightarrow \text{exp restOfListOfExp}$
- 73.  $\text{restOfListOfExp} \rightarrow \epsilon$
- 74.  $\text{restOfListOfExp} \rightarrow \text{COMMA exp restOfListOfExp}$

## 5.4. Acciones semánticas

- **program:** Crea una clase AST y lo asigna a la raíz

---

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

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

;
```

---

- **stmtlist:** puede crear una lista vacía o añadir una sentencia nueva a la lista ya existente. Si falla se creara una copia.

---

```
//interpreter.y
stmtlist: /* Empty: epsilon rule */
{
    // create an 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)->print();
            (*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;
}

;
```

---

## ■ stmt

---

```
//interpreter.y
stmt: SEMICOLON /* Empty statement: ";" */\
{
    // Create a new empty statement node
    $$ = new lp::EmptyStmt();
}
| asgn SEMICOLON
{
    // Default action
    // $$ = $1;
}
| ask_for_key SEMICOLON {}
| print SEMICOLON {}
| read SEMICOLON {}
| read_string SEMICOLON {}
| if SEMICOLON {}
| while SEMICOLON {}
| do_while SEMICOLON {}
| for SEMICOLON {}
| block {}
| placecursor SEMICOLON {}
| clear SEMICOLON {}
;
```

---

- **block:** Crea un bloque de nodos, no se utiliza en la practica porque utilizamos Stmtlist.

---

```
//interpreter.y
block: LETFCURLYBRACKET stmtlist RIGHTCURLYBRACKET
{
    $$ = new lp::BlockStmt($2);
}
;
```

---

- **controlSymbol:** Incrementa una variable de control para un correcto uso de los bucles.

---

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

---

- **if:** Tenemos dos versiones con sus respectivo control de errores, para un condicional simple y un condicional compuesto.

---

```
//interpreter.y
if: IF controlSymbol cond THEN stmtlist END_IF
{
    // Create a new if statement node
    $$ = new lp::IfStmt($3, $5);

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

```

}
| IF controlSymbol cond stmtlist END_IF
{
    execerror("Syntax error: missing ENTONCES symbol of if statement", "si cond
    ENTONCES ..");
}
| IF controlSymbol cond THEN stmtlist ELSE stmtlist END_IF
{
    $$ = new lp::IfStmt($3, $5, $7);
    control--;
}
| IF controlSymbol cond stmtlist ELSE stmtlist END_IF
{
    execerror("Syntax error: missing ENTONCES symbol of if statement", "si cond
    ENTONCES ..");
}
}
;

```

---

- **while:** Al igual que el condicional simple, genera nodos y disminuye la variable de control.

```

//interpreter.y
while: WHILE controlSymbol cond DO stmtlist END_WHILE
{
    $$ = new lp::WhileStmt($3, $5);
    control--;
}
| WHILE controlSymbol cond stmtlist END_WHILE
{
    execerror("Syntax error: missing HACER symbol of while statement", "mientras
    cond HACER ..");
}
;

```

---

- **do\_while:** Al igual que el condicional simple, genera nodos y disminuye la variable de control.

```

//interpreter.y
do_while: DO_WHILE controlSymbol stmtlist UNTIL cond
{
    $$ = new lp::DoWhileStmt($5, $3);
    control--;
}
| DO_WHILE controlSymbol stmtlist cond
{
    execerror("Syntax error: missing HASTA symbol of do while statement",
    "repetir stmtlist HASTA cond ..");
}
;

```

---

- **for:** Nuestro bucle para seria el mas completo ya que detectamos que no haya error en ninguna de sus variables.

```

//interpreter.y
for: FOR controlSymbol VARIABLE FROM exp UNTIL exp STEP exp DO stmtlist END_FOR
{
    $$ = new lp::ForStmt($3, $5, $7, $9, $11);
    control--;
}

```

```

}
| FOR controlSymbol VARIABLE FROM exp UNTIL exp DO stmtlist END_FOR
{
    $$ = new lp::ForStmt($3, $5, $7, $9);
    control--;
}
| FOR controlSymbol CONSTANT FROM exp UNTIL exp STEP exp DO stmtlist END_FOR
{
    execerror("Semantic error in for statement: it is not allowed to modify a constant", $3);
}
| FOR controlSymbol CONSTANT FROM exp UNTIL exp DO stmtlist END_FOR
{
    execerror("Semantic error in for statement: it is not allowed to modify a constant", $3);
}
| FOR controlSymbol VARIABLE exp UNTIL exp STEP exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing DESDE symbol of for statement", "para id DESDE ..");
}
| FOR controlSymbol VARIABLE FROM exp STEP exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing HASTA symbol of for statement", "para id DESDE exp HASTA ..");
}
| FOR controlSymbol VARIABLE FROM exp UNTIL exp STEP exp stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing HACER symbol of for statement", "para id DESDE exp HASTA exp PASO exp HACER ..");
}
| FOR controlSymbol VARIABLE exp UNTIL exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing DESDE symbol of for statement", "para id DESDE ..");
}
| FOR controlSymbol VARIABLE FROM exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing HASTA symbol of for statement", "para id DESDE exp HASTA ..");
}
| FOR controlSymbol VARIABLE FROM exp UNTIL exp stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing HACER symbol of for statement", "para id DESDE exp HASTA exp HACER ..");
}
| FOR controlSymbol VARIABLE FROM UNTIL exp STEP exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing from EXP argument of for statement", "para id desde EXP ..");
}
| FOR controlSymbol VARIABLE FROM exp UNTIL STEP exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing until EXP argument of for statement", "para id desde exp hasta EXP ..");
}
| FOR controlSymbol VARIABLE FROM UNTIL exp DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing from EXP argument of for statement", "para id desde EXP ..");
}

```

```

}
| FOR controlSymbol VARIABLE FROM exp UNTIL DO stmtlist END_FOR SEMICOLON
{
    execerror("Syntax error: missing until EXP argument of for statement", "para id
        desde exp hasta EXP ..");
}
;

```

---

- **switch:** Una opción para facilitar la elección entre múltiples ítems

```

//interpreter.y
switch: SWITCH controlSymbol LPAREN exp RPAREN listOfValues DEFAULT TWO_POINTS
    stmtlist END_SWITCH
{
    $$ = new lp::SwitchStmt($4, $6, $9);
    control --;
}
;

```

---

- **listOfValues:** Lista de los posibles casos del switch

```

//interpreter.y
listOfValues: VALUE NUMBER TWO_POINTS stmtlist restOfListOfValues
{
    $$ = $5;
    lp::ExpNode *e = new lp::NumberNode($2);
    lp::Value *v = new lp::Value(e, $4);
    $$->push_front(v);
}
| VALUE STRING TWO_POINTS stmtlist restOfListOfValues
{
    $$ = $5;
    lp::ExpNode *e = new lp::StringNode($2);
    lp::Value *v = new lp::Value(e, $4);
    $$->push_front(v);
}
;

```

---

- **restOfListOfValues:** Resto de los posibles casos del switch

```

//interpreter.y
restOfListOfValues: /*Empty: Epsilon rule*/
{
    $$ = new std::list<lp::Value *>();
}
| VALUE NUMBER TWO_POINTS stmtlist restOfListOfValues
{
    $$ = $5;
    lp::ExpNode *e = new lp::NumberNode($2);
    lp::Value *v = new lp::Value(e, $4);
    $$->push_front(v);
}
| VALUE STRING TWO_POINTS stmtlist restOfListOfValues
{
    $$ = $5;
    lp::ExpNode *e = new lp::StringNode($2);
}

```



```

        lp::Value *v = new lp::Value(e, $4);
        $$->push_front(v);
    }
;

```

---

- **cond:** Función simple que solo reconoce si esta entre paréntesis.

```

//interpreter.y
cond: LPAREN exp RPAREN
{
    $$ = $2;
}
;

```

---

- **asgn:** Función que asigna valores a la variable correspondiente.

```

//interpreter.y
asgn: VARIABLE ASSIGNMENT exp
{
    $$ = new lp::AssignmentStmt($1, $3);
}
| VARIABLE ASSIGNMENT asgn
{
    $$ = new lp::AssignmentStmt($1, (lp::AssignmentStmt *) $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);
}
;

```

---

- **ask\_for\_key:** Funcion para parar el programa cuando esta en modo interactivo.

```

//interpreter.y
ask_for_key: ASK_FOR_KEY LPAREN exp RPAREN
{
    $$ = new lp::AskForKeyStmt($3);
}
| ASK_FOR_KEY LPAREN RPAREN
{
    $$ = new lp::AskForKeyStmt(new lp::StringNode("Pulsa INTRO para
        continuar"));
}
;

```

---

- **print:** Función que muestra el contenido.

```

//interpreter.y
print: PRINT exp

```

```

    {
        $$ = new lp::PrintStmt($2);
    }
;

```

---

- **read:** Función que lee el contenido y lo almacena.

```

//interpreter.y
read: READ LPAREN VARIABLE RPAREN
    {
        $$ = new lp::ReadStmt($3);
    }
| READ LPAREN CONSTANT RPAREN
    {
        execerror("Semantic error in \"read statement\": it is not allowed to modify a
            constant ", $3);
    }
;

```

---

- **placecursor:** Función que crea un nodo place, que coloca nuestro curso en unas coordenadas.

```

placecursor: PLACECURSOR LPAREN exp COMMA exp RPAREN
    {
        $$ = new lp::PlaceStmt($3, $5);
    }
;

```

---

- **clear:** Función que crea un nodo clear, que limpia la pantalla.

```

clear: CLEAR
    {
        $$ = new lp::ClearStmt();
    }
;

```

---

- **read\_string:** Función que lee variables tipo cadena.

```

read_string: READ_STRING LPAREN VARIABLE RPAREN
    {
        $$ = new lp::ReadStmtString($3);
    }
| READ_STRING LPAREN CONSTANT RPAREN
    {
        execerror("Semantic error in \"read string statement\": it is not allowed
            to modify a constant ", $3);
    }
;

```

---

- **exp** Son todas las acciones semánticas que puede hacer un nodo Exp con los operadores.

```

exp: NUMBER
    { $$ = new lp::NumberNode($1); }
| STRING
    { $$ = new lp::StringNode($1); }

```

```

| exp PLUS exp
{ $$ = new lp::PlusNode($1, $3); }
| exp MINUS exp
{ $$ = new lp::MinusNode($1, $3); }
| exp MULTIPLICATION exp
{ $$ = new lp::MultiplicationNode($1, $3); }
| exp DIVISION exp
{ $$ = new lp::DivisionNode($1, $3); }
| exp INTEGER_DIVISION exp
{ $$ = new lp::IntegerDivisionNode($1, $3); }
| exp CONCATENATION exp
{ $$ = new lp::ConcatenationNode($1, $3); }
| LPAREN exp RPAREN
{ $$ = $2; }
| PLUS exp %prec UNARY
{ $$ = new lp::UnaryPlusNode($2); }
| MINUS exp %prec UNARY
{ $$ = new lp::UnaryMinusNode($2); }
| exp MODULO exp
{ $$ = new lp::ModuloNode($1, $3); }
| exp POWER exp
{ $$ = new lp::PowerNode($1, $3); }
| VARIABLE
{ $$ = new lp::VariableNode($1); }
| CONSTANT
{ $$ = new lp::ConstantNode($1); }
| exp GREATER_THAN exp
{ $$ = new lp::GreaterThanNode($1,$3); }
| exp GREATER_OR_EQUAL exp
{ $$ = new lp::GreaterOrEqualNode($1,$3); }
| exp LESS_THAN exp
{ $$ = new lp::LessThanNode($1,$3); }
| exp LESS_OR_EQUAL exp
{ $$ = new lp::LessOrEqualNode($1,$3); }
| exp EQUAL exp
{ $$ = new lp::EqualNode($1,$3); }
| exp NOT_EQUAL exp
{ $$ = new lp::NotEqualNode($1,$3); }
| exp AND exp
{ $$ = new lp::AndNode($1,$3); }
| exp OR exp
{ $$ = new lp::OrNode($1,$3); }
| NOT exp
{ $$ = new lp::NotNode($2); }
| 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);
}
;

```

---

- **listOfExp:** Crea una lista vacía de expresiones, si ya existe añade la expresión.

```

listOfExp: /* Empty list of numeric expressions */
{
    $$ = new std::list<lp::ExpNode *>();
}
| exp restOfListOfExp
{
    $$ = $2;
    $$->push_front($1);
}
;

```

---

- **restOfListOfExp:** Funciona igual que listOfExp pero trabaja con una coma.

```

restOfListOfExp: /* Empty list of numeric expressions */
{
    $$ = new std::list<lp::ExpNode *>();
}
| COMMA exp restOfListOfExp
{
    // Get the list of expressions
    $$ = $3;

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

```

;

}

---

# Árbol sintáctico abstracto

Dentro de la jerarquía del árbol tenemos las siguientes clases:

## ■ Clase **Statement**

- AskForKeyStmt: Clase que permite interrumpir la ejecución del programa cuando es en modo interactivo.
- AssignmentStmt: Clase que evalúa la función de asignación.
- DowhileStmt: Clase que evalúa la función del bucle **repetir**.
- EmptyStmt: Clase que evalúa una función vacía.
- ForStmt: Clase que evalúa la función del bucle **para**.
- SwitchStmt: Clase que evalúa la función de la sentencia **switch**.
- IfStmt: Clase que evalúa la función **si**.
- PrintStmt: Clase que evalúa la función **escribir**.
- ReadStmt: Clase que evalúa la función **leer**.
- ReadStringStmt: Clase que evalúa la función **escribir\_cadena**.
- WhileStmt: Clase que evalúa la función del bucle **mientras**.

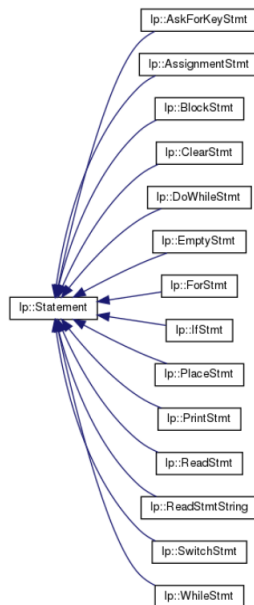


Figura 6.1: Estructura de la clase Statement

## ■ Clase **ExpNode**

- **BuiltinFunctionNode**: Clase que define los constructores de las funciones predefinidas.
  - **BuiltinFunctionNode\_0**: Clase que define los constructores de las funciones sin parámetros.
  - **BuiltinFunctionNode\_1**: Clase que define los constructores de las funciones con un único parámetro.
  - **BuiltinFunctionNode\_2**: Clase que define los constructores de las funciones con dos parámetros.
- **ConstantNode**: Clase que define las constantes en la tabla de símbolos.
- **NumberNode**: Clase que define las variables numéricas dentro de la tabla de símbolos.
- **OperatorNode**: Clase que define los operadores de las variables.
  - **LogicalOperatorNode**: Clase que define los operadores lógicos que afectan a las variables.
    - ◊ **AndNode**: Clase que define el operador lógico AND.
    - ◊ **OrNode**: Clase que define el operador lógico OR.
  - **NumericOperatorNode**: Clase que define los operadores para las variables numéricas.
    - ◊ **DivisionNode**: Clase que define la operación de división entre dos variables numéricas.
    - ◊ **IntegerDivisionNode**: Clase que define la operación de división entera entre dos variables numéricas.
    - ◊ **MinusNode**: Clase que define la operación de resta entre dos variables numéricas.
    - ◊ **ModuloNode**: Clase que define la operación de módulo entre dos variables numéricas.
    - ◊ **MultiplicationNode**: Clase que define la operación de multiplicación entre dos variables numéricas.
    - ◊ **PlusNode**: Clase que define la operación de suma entre dos variables numéricas.
    - ◊ **PowerNode**: Clase que define la operación de potencia entre dos variables numéricas.
  - **RelationalOperatorNode**: Clase que define los operadores para las variables numéricas o alfanuméricas que dan como resultado verdadero o falso.
    - ◊ **EqualNode**: Clase que define la operación de igualdad para las variables numéricas o alfanuméricas.
    - ◊ **GreterOrEqualNode**: Clase que define la operación de mayor o igual para las variables numéricas o alfanuméricas.
    - ◊ **GreterThanlNode**: Clase que define la operación de mayor que para las variables numéricas o alfanuméricas.
    - ◊ **LessrOrEqualNode**: Clase que define la operación de menor o igual que para las variables numéricas o alfanuméricas.
    - ◊ **LessThanlNode**: Clase que define la operación de menor que para las variables numéricas o alfanuméricas.
    - ◊ **NotEqualNode**: Clase que define la operación de desigualdad para las variables numéricas o alfanuméricas.
  - **StringOperatorNode**: Clase que define los operadores específicos para las variables alfanuméricas.
    - ◊ **ConcatenationNode**: Clase que define la función de concatenación para dos variables alfanuméricas.
- **StringNode**: Clase que define las variables cadena dentro de la tabla de símbolos.
- **UnaryOperatorNode**: Clase que define los operadores unarios que afectan a las variables.
  - **LogicalUnaryOperatorNode**: Clase que define los operadores unarios lógicos.
    - ◊ **NotNode**: Clase que define el operador NOT.
  - **NumericUnaryOperatorNode**: Clase que define los operadores unarios que afectan a las variables numéricas.

- ◊ UnaryMinusNode: Clase que define el operador unario -
- ◊ UnaryPlusNode: Clase que define el operador unario +
- VariableNode: Clase que define a las variables que se generaran en la tabla de símbolos.

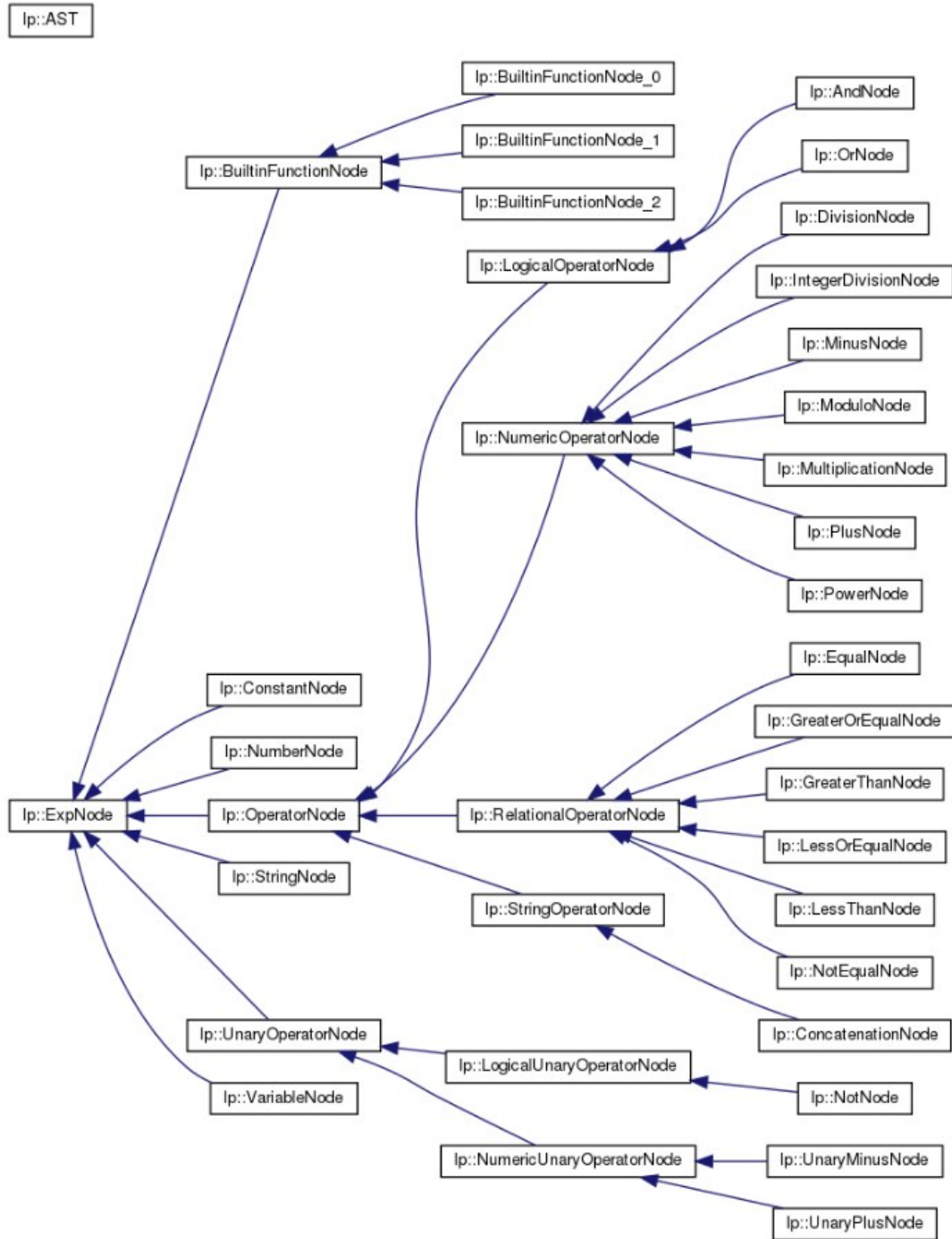


Figura 6.2: Estructura de la clase ExpNode



# Modo de obtención del intérprete

Ahora se hará un desglose con los directorios y los ficheros del proyecto.

- ast: Subdirectorio con los ficheros del árbol de sintaxis abstracto y las clases relacionadas.
  - ast.cpp: Fichero donde codificaremos las clases del árbol de sintaxis abstracta.
  - ast.hpp: Fichero de cabecera donde declararemos las clases del árbol de sintaxis abstracta.
  - makefile: Fichero para la compilación del subdirectorio ast.
- error: Subdirectorio con los ficheros para el tratamiento de errores.
  - error.cpp: Fichero donde codificaremos las funciones de recuperación de error.
  - error.hpp: Fichero de cabecera donde declararemos las funciones de recuperación de error.
  - makefile: Fichero para la compilación del subdirectorio error.
- examples: Subdirectorio con ficheros de prueba.
  - test: Fichero de ejemplo sin errores
  - test2: Fichero de ejemplo sin errores
  - test-error: Fichero de ejemplo con errores
- includes: Subdirectorio con ficheros para la mejora del programa.
  - macros.hpp: Macros para mejorar la visualización por pantalla.
- parser: Subdirectorio con ficheros de para el análisis léxico y sintáctico.
  - interpreter.l: Fichero de lex con las expresiones regulares del analizador léxico.
  - interpreter.y: Fichero de yacc con la gramática del analizador sintáctico.
  - makefile: Fichero para la compilación del subdirectorio parser.
- Doxyfile: Fichero que genera la documentación de los subdirectorios del proyecto.
- interpreter.cpp: Programa principal de nuestro proyecto.
- makefile: Fichero para la compilación del intérprete.

- table: Subdirectorio con los ficheros relacionados con la tabla de símbolos.
  - builtin.cpp: Fichero donde codificaremos las funciones de creación de funciones predefinidas.
  - builtin.hpp: Fichero de cabecera donde declararemos las clases de builtin.
  - builtinParameter0.cpp: Fichero donde codificaremos la función predefinida builtinParameter0.
  - builtinParameter0.hpp: Fichero de cabecera donde declararemos las clases de builtinParameter0.
  - builtinParameter1.cpp: Fichero donde codificaremos la función predefinida builtinParameter1.
  - builtinParameter1.hpp: Fichero de cabecera donde declararemos las clases de builtinParameter1.
  - builtinParameter2.cpp: Fichero donde codificaremos la función predefinida builtinParameter2.
  - builtinParameter2.hpp: Fichero de cabecera donde declararemos las clases de builtinParameter2.
  - constant.cpp: Fichero donde codificaremos las funciones de la clase constant, que hereda de la clase symbol.
  - constant.hpp: Fichero de cabecera donde declararemos las clases de constant.
  - init.cpp: Fichero donde codificaremos las funciones que se inicializan en la tabla de símbolos con las constantes predefinidas.
  - init.hpp: Fichero de cabecera donde declararemos las clases de init.
  - keyword.cpp: Fichero donde codificaremos la función de la clase keyword.
  - keyword.hpp: Fichero de cabecera donde declararemos las clases de keyword, que hereda de la clase Symbol.
  - logicalConstant.cpp: Fichero donde codificaremos las funciones de la clase logicalConstant.
  - logicalConstant.hpp: Fichero de cabecera donde declararemos la clase de logicalConstant.
  - logicalVariable.cpp: Fichero donde codificaremos las funciones de la clase logicalVariable.
  - logicalVariable.hpp: Fichero de cabecera donde declararemos la clase de logicalVariable.
  - mathFunction.cpp: Fichero donde codificaremos las funciones matemáticas predefinidas.
  - mathFunction.hpp: Fichero de cabecera donde declararemos las funciones matemáticas predefinidas.
  - numericConstant.cpp: Fichero donde codificaremos las funciones de la clase numericConstant.
  - numericConstant.hpp: Fichero de cabecera donde declararemos la clase de numericConstant.
  - numericVariable.cpp: Fichero donde codificaremos las funciones de la clase numericVariable.
  - numericVariable.hpp: Fichero de cabecera donde declararemos la clase de numericVariable.
  - stringVariable.cpp: Fichero donde codificaremos las funciones de la clase stringVariable.
  - stringVariable.hpp: Fichero de cabecera donde declararemos la clase de stringVariable.
  - symbol.cpp: Fichero donde codificaremos las funciones de la clase symbol.
  - symbol.hpp: Fichero de cabecera donde declararemos la clase de symbol.
  - table.cpp: Fichero donde codificaremos las funciones de la clase table.
  - table.hpp: Fichero de cabecera donde declararemos la clase de table.
  - tableInterface.hpp: Fichero donde se define la clase abstracta tableInterface.
  - variable.cpp: Fichero donde codificaremos la función de la clase variable.
  - variable.hpp: Fichero de cabecera donde declararemos las clases de variable, que hereda de la clase Symbol.

Árbol de los directorios y sus ficheros, dentro del subdirectorio table faltan ficheros para reducir el tamaño del árbol.

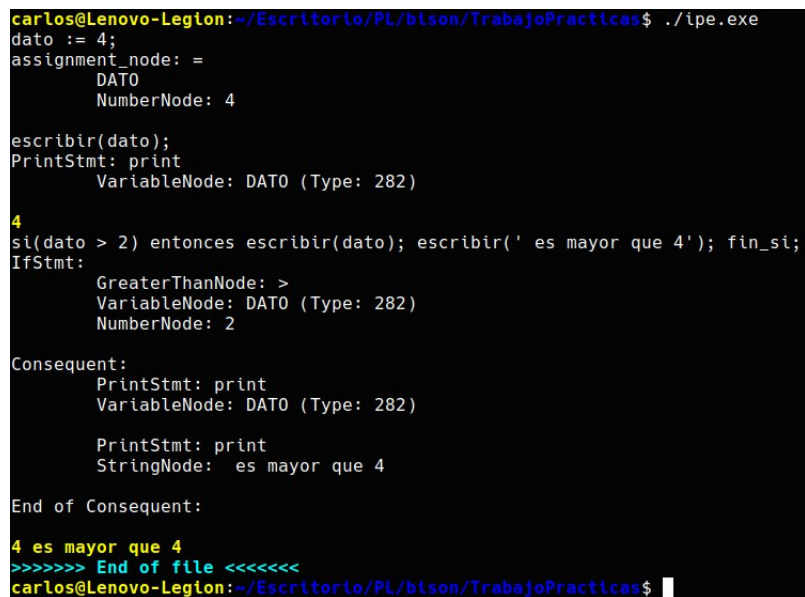
```
TrabajoPracticas
├── ast
│   ├── ast.cpp
│   ├── ast.hpp
│   └── makefile
├── error
│   ├── error.cpp
│   ├── error.hpp
│   └── makefile
├── exmaples
│   ├── test
│   ├── test2
│   └── test-error
├── includes
│   └── macros.hpp
├── parser
│   ├── interpreter.l
│   ├── interpreter.y
│   └── makefile
├── table
│   ├── init.cpp
│   ├── init.hpp
│   ├── keyword.cpp
│   ├── keyword.hpp
│   ├── logicalConstant.cpp
│   ├── logicalConstant.hpp
│   ├── logicalVariable.cpp
│   ├── logicalVariable.hpp
│   ├── mathFunction.cpp
│   ├── mathFunction.hpp
│   ├── numericConstant.cpp
│   ├── numericConstant.hpp
│   ├── numericVariable.cpp
│   ├── numericVariable.hpp
│   ├── stringVariable.cpp
│   ├── stringVariable.hpp
│   ├── symbol.cpp
│   ├── symbol.hpp
│   ├── symbolInterface.cpp
│   ├── symbolInterface.hpp
│   ├── table.cpp
│   ├── table.hpp
│   ├── tableInterface.hpp
│   ├── variable.cpp
│   ├── variable.hpp
│   └── ...
├── Doxyfile
├── interpreter.cpp
└── makefile
```

# Modo de ejecución

Nuestro programa se podrá ejecutar de distintas formas, desde la forma interactiva podremos interactuar mientras se ejecuta nuestro programa o sino podremos ejecutarlo pasando como argumento un documento con la extensión ".e".

## 8.1. Ejecución interactiva

El modo de ejecución interactiva permite al usuario hacer uso del programa desde la terminal, gracias a este modo podremos ir escribiendo nuestro código en tiempo de ejecución y también nos permite interrumpir el programa en mitad de una ejecución.



```
carlos@Lenovo-Legion:~/Escritorio/PL/bison/TrabajoPracticas$ ./ipe.exe
dato := 4;
assignment_node: =
    DATO
    NumberNode: 4

escribir(dato);
PrintStmt: print
    VariableNode: DATO (Type: 282)

4
si(dato > 2) entonces escribir(dato); escribir(' es mayor que 4'); fin_si;
IfStmt:
    GreaterThanNode: >
    VariableNode: DATO (Type: 282)
    NumberNode: 2

Consequent:
    PrintStmt: print
    VariableNode: DATO (Type: 282)

    PrintStmt: print
    StringNode: es mayor que 4

End of Consequent:

4 es mayor que 4
>>>>>> End of file <<<<<<
carlos@Lenovo-Legion:~/Escritorio/PL/bison/TrabajoPracticas$
```

Figura 8.1: Ejecución interactiva del intérprete

## 8.2. Ejecución a partir de un fichero

Desde este modo el usuario podrá pasar como argumento un fichero el cual ya tendrá el pseudocódigo que queremos ejecutar, esto nos aporta gran utilidad a la hora de utilizar el intérprete pasándole directamente documentos más complejos.

Para ejecutar nuestro programa seria de la siguiente manera: `./ipe.exe [fichero.e]`



```
carlos@Lenovo-Legion:~/Escritorio/PL/blison/TrabajoPracticas$ ./ipe.exe examples/factorial.e
>>>>>> End of file <<<<<<
Introduce un numero: 6
El factorial de 6 es 720
carlos@Lenovo-Legion:~/Escritorio/PL/blison/TrabajoPracticas$
```

Figura 8.2: Ejecución a partir del fichero factorial.e del intérprete

# Ejemplos

A continuación se muestran los ejemplos utilizados en nuestro intérprete y una breve descripción de su función.

## 9.1. conversion.e

---

```
<<
Asignatura:  Procesadores de Lenguajes

    Titulacin:  Ingeniera Informtica
    Especialidad: Computacin
    Curso:      Tercero
    Cuatrimestre: Segundo

    Departamento: Informtica y Anlisis Numrico
    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('\nIntroduce 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');
```

---

Uno de los ejemplos propuestos por el profesor, prueba el cambio de tipo de variable de numero a cadena

## 9.2. ecuacion.e

---

```
<<
Asignatura:   Procesadores de Lenguajes

Titulacin:    Ingeniera Informtica
Especialidad: Computacin
Curso:        Tercero
Cuatrimestre: Segundo

Departamento: Informtica y Anlisis Numrico
Centro:       Escuela Politecnica Superior de Crdoba
Universidad de Crdoba

Curso acadmico: 2020 - 2021
>>

@ Este fichero pide los argumentos de una ecuacion de segundo grado y la resuelve

a := b := c := 0;
res1 := 'no definido';
res2 := 'no definido';

escribir_cadena('Vamos a resolver una ecuacion de segundo grado\n');
escribir_cadena('Introduce el coeficiente a: ');
leer(a);

escribir_cadena('Introduce el coeficiente b: ');
leer(b);

escribir_cadena('Introduce el coeficiente c: ');
leer(c);

si(a <> 0 #y b <> 0 #y c <> 0) entonces
    raiz := sqrt(b**2 - 4 * a * c);
    res1 := (-b + raiz) / 2 * a;
    res2 := (-b - raiz) / 2 * a;
si_no
    si(a = 0 #y b <> 0 #y c <> 0) entonces
        res1 := -c / b;
    si_no
        si(a <> 0 #y b = 0 #y c <> 0) entonces
            raiz := sqrt(-c / a);
            res1 := raiz;
            res2 := -raiz;
        si_no
            si(a <> 0 #y b <> 0 #y c = 0) entonces
                res1 := 0;
                res2 := -b / a;
            si_no
                res1 := '0 o indefinido';
                escribir('\n0 el resultado es 0 o no tiene solucion\n\n');
            fin_si;
    fin_si;
```

```

        fin_si;
    fin_si;
fin_si;

escribir('Dada la ecuacion: ');
escribir(a);
escribir('x^2');

si(b >= 0) entonces
    escribir('+');
fin_si;

escribir(b);
escribir('x');

si(c >= 0) entonces
    escribir('+');
fin_si;

escribir(c);
escribir('=0;\n' || 'Sus resultados son:\n');
escribir('x1 = ');
escribir(res1);
escribir('\nx2 = ');
escribir(res2);

escribir('\n');

```

---

Este ejemplo utiliza los operadores relacionales, la asignación múltiple, la concatenación y la interacción con el usuario para resolver una ecuación.

### 9.3. ecuacion-error.e

---

```

<<
Asignatura:  Procesadores de Lenguajes

Titulacin:   Ingeniera Informtica
Especialidad: Computacin
Curso:       Tercero
Cuatrimestre: Segundo

Departamento: Informtica y Anlisis Numrico
Centro:       Escuela Politcnica Superior de Crdoba
Universidad de Crdoba

Curso acadmico: 2020 - 2021
>>

@ Este fichero pide los argumentos de una ecuacion de segundo grado y la resuelve

a := b = c := 0;
res1 := 'no definido'fd;
res__2 := 'no definido';

escribir_cadena'Vamos a resolver una ecuacion de segundo grado\n');

```



```

escribir_cadena('Introduce el coeficiente a: ');
leer(A);

escribir_cadena('Introduce el coeficiente b: ');
leer(b);

escribir_cadena('Introduce el coeficiente c: ');
leer(c);

si(a <> 0 #y b <> 0 #y c <> 0) entonces
    raiz := sqrt(b**2 - 4a * a * c);
    res1 := (-b + raiz) / 2 * a;
    res2 := (-b - raiz) / 2 * a;
si_no
    si(a = 0 #y b <> 0 #y c <> 0) entonces
        res1 := -c / b;
    si_no
        si(a <> 0 #y b = 0 #y c <> 0) entonces
            raiz := sqrt(-c / a);
            res1 := raiz;
            res2 := -raiz;
        si_no
            si(a <> 0 #y b <> 0 #y c = 0) entonces
                res1 := 0;
                res2 := -b / a;
            si_no
                res1 := '0 o indefinido';
                escribir('\n0 el resultado es 0 o no tiene solucion\n\n');
            fin_si;
        fin_si;
    fin_si;
fin_si;

escribir('Dada la ecuacion: ');
#escribir(a);
escribir('x^2');

si(_b >= 0)
    escribir('+');
fin_si;

escribir(b);
escribir('x');

si(c >= 'a') entonces
    escribir('+');
fin_si;

escribir(c);
escribir('=0;\n' || 'Sus resultados son:\n');
escribir('x1 = ');
escribir(res1)
escribir('\nx2 = ');
escribir(res2);

escribir('\n');

```

---

Mismo fichero que el anterior, pero con algunos errores.

## 9.4. entrada.txt

---

```
    si (-PI> 0) entonces escribir verdadero; fin_si;

dato := 2;
escribir dato;

escribir('\n');

si (dato > 0) entonces escribir dato; si_no escribir -dato; fin_si;

escribir '\n';
```

---

Este fichero no tiene la extensión '.e', por lo que debería lanzar un aviso.

## 9.5. factorial.e

---

```
    escribir_cadena('Introduce un numero: ');
leer(dato);

escribir_cadena('El factorial de ');
escribir(dato);

si (dato < 0) entonces
    f := 0 ;
si_no
    i := dato;
    f := 1;

    mientras(i>1) hacer
        f := f * i;
        i := i - 1;
    fin_mientras;
fin_si;

escribir_cadena(' es ');
escribir(f);
escribir_cadena('\n');
```

---

Este fichero calcula el factorial de un número introducido por teclado. Utiliza la sentencia mientras.

## 9.6. factorial-error.e

---

```
    escribir_cadena('Introduce un numero: ');
leer(dato);

escribir dato;
escribir '\n';
```

---

```

si (dato < 0) entonces
    f := 0
si_no
    i := dato;
    f := 1;
    mientras i>1) hacer
        f := f * i;
        i := i - 1;
    fin_mientras;
fin_si

escribir f;
escribir '\n';

```

---

Este es el mismo fichero que el anterior, pero con algunos errores.

## 9.7. menu.e

---

```

<<
Asignatura:  Procesadores de Lenguajes

Titulacin:   Ingeniera Informtica
Especialidad: Computacin
Curso:       Tercero
Cuatrimestre: Segundo

Departamento: Informtica y Anlisis Numrico
Centro:       Escuela Politecnica Superior de Crdoba
Universidad de Crdoba

Curso acadmico: 2020 - 2021

Fichero de ejemplo para el intrprete de pseudocodigo en espaol: 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 intrprete de pseudocodigo en espaol:\'ipe.exe\'.');

#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 nmero --> 1 ');

#lugar(11,10);
escribir_cadena(' Mximo comn 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 intrprete ipe.exe ');

@ Factorial de un nmero
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);

            @ Mximo comn divisor
si_no

```

```

si (opcion = 2)
    entonces
        #lugar(10,10);
        escribir_cadena(' Mximo comn divisor de dos nmeros ');

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

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

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

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

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

    @ Se aplica el mtodo 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(' Mximo comn 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\n');

```

---

Este fichero usa un menu para realizar diversas funciones. Utiliza todas las sentencias excepto el switch. La variedad de casos de implementa con múltiples ifs.

## 9.8. menuCasos.e

---

```

@ Ejemplo de menu con sentencia CASOS

repetir

@ Opciones disponibles

escribir_cadena(' OPCIONES\n');
escribir_cadena(' OPCION 1 ----> 1\n');
escribir_cadena(' OPCION 2 ----> 2\n');
escribir_cadena(' OPCION 3 ----> 3\n');
escribir_cadena(' FINALIZAR ---> 4\n');
escribir_cadena(' Elige una opcion: ');

leer(opcion);

casos(opcion)
    valor 1: escribir_cadena('\n Has elegido la opcion 1\n');
    valor 2: escribir_cadena('\n Has elegido la opcion 2\n');
    valor 3: escribir_cadena('\n Has elegido la opcion 3\n');
    valor 4: escribir_cadena('\n Que tenga un buen da\n');
    defecto:
        escribir_cadena('\n La opcion no es valida\n');
        escribir_cadena(' Prueba de nuevo\n');
fin_casos;

escribir_cadena('\n Pulse una tecla para continuar --> ');
leer_cadena(pausa);

#borrar;

hasta (opcion = 4);

```

---

Este fichero utiliza la sentencia CASOS para implementar un pequeño menú.

## 9.9. op-aritmeticas.e

---

```

datoNumero := 0;
escribir_cadena('ASIGNACION\n');

```

```

escribir(datoNumero);

datoNumero := datoNumero + 2;
escribir_cadena('\nSUMA CON NUMBER\n');
escribir(datoNumero);

datoNumero := 4 + 5;
escribir_cadena('\nSUMA DE NUMBERS\n');
escribir(datoNumero);

datoAuxiliar := 2;

datoNumero := datoNumero - datoAuxiliar;
escribir_cadena('\nRESTA DE VARIABLES\n');
escribir(datoNumero);
escribir(datoAuxiliar);

datoNumero := 34 - datoNumero;
escribir_cadena('\nNUMER - VARIABLE\n');
escribir(datoNumero);

datoNumero := 10;

datoNumero := datoNumero * datoAuxiliar;
escribir_cadena('\nMULTIPLICACION DE VARIABLES\n');
escribir(datoNumero);

datoNumero := datoNumero * 0.5;
escribir_cadena('\nVARIABLE * NUMBER\n');
escribir(datoNumero);

datoNumero := 10 / 5;
escribir_cadena('\nDIVISION DE NUMBERS\n');
escribir(datoNumero);

datoNumero := datoNumero / 0.5;
escribir_cadena('\nVARIABLE / NUMBER\n');
escribir(datoNumero);

datoNumero := 5;

datoNumero := datoNumero #div datoAuxiliar;
escribir_cadena('\nDIVISION ENTERA\n');
escribir(datoNumero);

datoNumero := 5;

datoNumero := datoNumero #div 4;
escribir_cadena('\nVARIABLE #div NUMBER\n');
escribir(datoNumero);

datoNumero := datoNumero #mod datoAuxiliar;
escribir_cadena('\nMODULO\n');
escribir(datoNumero);

datoNumero := 5;

datoNumero := datoNumero #mod 3;

```

```
escribir_cadena('\nVARIABLE #mod NUMBER\n');
escribir(datoNumero);
escribir_cadena('\n');
```

---

Este fichero no tiene la extensión '.e', por lo que debería lanzar un aviso.

## 9.10. op-relacionales.e

---

```
dato1 := verdadero;
dato2 := falso;

si(dato1 #0 dato2) entonces
    escribir(dato1);
    escribir(' o ');
    escribir(dato2);
    escribir(' es igual a verdadero\n');
si_no
    escribir(dato1);
    escribir(' o ');
    escribir(dato2);
    escribir(' es igual a falso\n');
fin_si;

escribir('Aunque dato2');
escribir(' sea ');
escribir(dato2);
escribir(', si lo negamos es ');
escribir(#NO DATO2);
escribir('\n');
```

---

Este contiene operaciones con los operadores relacionales del intérprete.

## 9.11. op-relacionales-error.e

---

```
dato1 := verdadero
dato2 := falso;

si(dato1 #0 dato2 entonces
    escribir(dato1);
    escribir(' o ');
    escribir(dato2);
    escribir(' es igual a verdadero\n');
si_no
    escribir(dato1);
    escribir(' o ');
    escribir(dato2);
    escribir(' es igual a falso\n');
fin_si

escribir('Aunque dato2');
escribir(' sea ');
escribir(dato2)
escribir(', si lo negamos es ');
escribir(#NOO DATO2);
```



```
#escribir('\n');
```

---

Este fichero es igual que el anterior pero contiene algunos errores.

## 9.12. primo.e

---

```
<<
Asignatura:  Procesadores de Lenguajes

Titulacin:   Ingeniera Informtica
Especialidad: Computacin
Curso:       Tercero
Cuatrimestre: Segundo

Departamento: Informtica y Anlisis Numrico
Centro:      Escuela Politcnica Superior de Crdoba
Universidad de Crdoba

Curso acadmico: 2020 - 2021

Este ejemplo determina si un numero introducido por el usuario es primo o no
>>

escribir('\n\tIntroduce un numero: ');
leer(numero);

num_divisores := 0;

para i desde 1 hasta numero+1 hacer
    si (numero #mod i = 0) entonces
        num_divisores := num_divisores + 1;
    fin_si;
fin_para;

escribir '\n\tEl ';
escribir(NUMERO);
si(num_divisores = 2) entonces
    escribir(' es primo\n\n');
si_no escribir(' no es primo\n\n');
fin_si;
```

---

Este fichero calcula si un número introducido por el usuario es primo o no. Utiliza el operador mod

## 9.13. primo-error.e

---

```
<<
Asignatura:  Procesadores de Lenguajes

Titulacin:   Ingeniera Informtica
Especialidad: Computacin
Curso:       Tercero
Cuatrimestre: Segundo

Departamento: Informtica y Anlisis Numrico
```

Centro: Escuela Politecnica Superior de Crdoba  
Universidad de Crdoba

Curso academico: 2020 - 2021

Este ejemplo determina si un numero introducido por el usuario es primo o no  
>>

```
escribir('\n\tIntroduce un numero: ')
leer(numero_);

num_divisores = 0;

para i desde hasta numero+1 hacer
    si (numero #mod i = 0) entonces
        num_divisores := num_divisores + 1;
    fin_si;
fin_para;

escribir '\n\tEl ';
escribir(NUMERO);
si(num_divisores = 2)
    escribir(' es primo\n\n');
si_no escribir(' no es primo\n\n');
fin_si
```

---

Este fichero es igual que el anterior pero contiene algunos errores.

## 9.14. test2.txt

---

```
a := b := PI;

escribir(a);
escribir_cadena('\n');
escribir(b);
escribir_cadena('\n');
escribir(PI);
escribir_cadena('\n');

dato := 2;
escribir(dato);
escribir_cadena('\n');

dato := verdadero;

escribir(dato);
escribir_cadena('\n');

dato := 9;
escribir(dato);
escribir_cadena('\n');

escribir(3>PI);
escribir_cadena('\n');
```

---

Este fichero no tiene la extensión '.e' por lo que debería lanzar un aviso. El ejemplo únicamente realiza unas sencillas operaciones de entrada y de salida.

# Conclusiones

## 10.1. Reflexión general

Una vez finalizada el trabajo, hemos visto que el tema de crear un intérprete con una buena funcionalidad es abrumador, gracias a los documentos de las practicas y a la modularización de los componentes se hizo ameno la construcción, pero es un trabajo con mucho esfuerzo y muchas horas de trabajo.

## 10.2. Puntos fuertes y puntos débiles del intérprete

### 10.2.1. Puntos fuertes

El intérprete cumple con su función y hemos acabado muy contentos con el trabajo, nos gusta bastante el cambio de poder escribir una variable con una única función gracias a que reconoce si es alfanumérica o numérica antes de generarla. Por la general ha sido un gran trabajo.

### 10.2.2. Puntos débiles

El mayor problema del intérprete ha sido no dedicarle el suficiente tiempo ya que hemos podido implementar lo que se nos pedía, no hemos generado funciones complementarias.

# Bibliografía

- [1] *The Lex & Yacc Page*.  
Disponible en: <http://dinosaur.compilertools.net>
- [2] El sistema operativo GNU. 2014. *GNU Bison*. [Consulta: 2 de junio de 2021].  
Disponible en: <https://www.gnu.org/software/bison/>
- [3] Fernández García, Nicolás Luis. 2021. *Procesadores de lenguajes*. [Sitio web]. Universidad de Córdoba  
Disponible en: <https://moodle.uco.es/m2021/course/view.php?id=2078>