

# INTÉRPRETE EN PSEUDOCÓDIGO EN ESPAÑOL: IPE

## **NOMBRE Y APELLIDOS DE LOS AUTORES**

Jaime Lorenzo Sánchez

Francisco Cruceira Lloret

## **NOMBRE DE LA ASIGNATURA**

Procesadores de Lenguajes

## **NOMBRE DE LA TITULACIÓN**

Grado de Ingeniería Informática-Computación

## **CURSO**

Tercer curso

Segundo Cuatrimestre

## **CURSO ACADÉMICO**

2020-2021

Escuela Politécnica Superior de Córdoba-Universidad de Córdoba

## **LUGAR**

Córdoba

11 de febrero de 2022



# Índice general

<b>1. INTRODUCCIÓN</b>	<b>1</b>
<b>2. LENGUAJE DEL PSEUDOCÓDIGO</b>	<b>3</b>
2.1. COMPONENTES LÉXICOS . . . . .	3
2.2. SENTENCIAS . . . . .	7
<b>3. TABLA DE SÍMBOLOS</b>	<b>12</b>
<b>4. ANÁLISIS LÉXICO</b>	<b>18</b>
<b>5. ANÁLISIS SINTÁCTICO</b>	<b>25</b>
5.1. SÍMBOLOS DE LA GRAMÁTICA . . . . .	25
5.1.1. SÍMBOLOS TERMINALES (COMPONENTES LÉXICOS) . . . . .	25
5.1.2. SÍMBOLOS NO TERMINALES . . . . .	28
5.2. REGLAS DE PRODUCCIÓN DE LA GRAMÁTICA . . . . .	29
5.3. ACCIONES SEMÁNTICAS . . . . .	30
<b>6. CÓDIGO DE AST</b>	<b>32</b>
<b>7. FUNCIONES AUXILIARES</b>	<b>39</b>
<b>8. MODO DE OBTENCIÓN DEL INTÉRPRETE</b>	<b>41</b>
8.1. NOMBRE Y DESCRIPCIÓN DE CADA DIRECTORIO . . . . .	41
8.2. NOMBRE Y DESCRIPCIÓN DE CADA FICHERO UTILIZADO DE CA- DA DIRECTORIO . . . . .	42
<b>9. MODO DE EJECUCIÓN DEL INTÉRPRETE</b>	<b>46</b>

9.1. INTERACTIVA . . . . .	46
9.2. A PARTIR DE UN FICHERO . . . . .	46
<b>10.EJEMPLOS</b>	<b>48</b>
<b>11.CONCLUSIONES</b>	<b>56</b>
11.1. PUNTOS FUERTES Y PUNTOS DÉBILES DEL INTÉRPRETE DESA- ROLLADO . . . . .	56
11.2. REFLEXIÓN FINAL SOBRE EL TRABAJO REALIZADO . . . . .	56
<b>12.BIBLIOGRAFÍA</b>	<b>58</b>
<b>13.ANEXOS</b>	<b>59</b>

# Capítulo 1

## INTRODUCCIÓN

Se explicará una breve descripción del trabajo realizado y de las partes del documento.

El trabajo realizado consiste en utilizar Flex y Bison para diseñar un analizador de lenguajes, el cual convierte la descripción formal de un lenguaje escrito como una gramática de contexto libre LALR en un programa que realiza análisis sintácticos.

Las partes del documento son las siguientes:

1. Lenguaje del pseudocódigo: Se explicarán las características del lenguaje de pseudocódigo, dando una pequeña explicación de los componentes léxicos y sentencias diseñadas.
2. Tabla de Símbolos: Se realizará una pequeña descripción de las clases utilizadas por el analizador, utilizando figuras generadas por doxygen.
3. Análisis Léxico: Se realizará una descripción de los componentes léxicos y sus expresiones regulares utilizadas por el analizador.
4. Análisis Sintáctico: Se realizará una pequeña descripción de la gramática de contexto libre del lenguaje de la gramática, en el cual se explicará los símbolos de la gramática (componentes léxicos y símbolos no terminales), las reglas de producción de la gramática y las acciones semánticas.

5. Código de AST: Se realizará un pequeño resumen y una breve descripción de las clases utilizadas, utilizando las figuras generadas por doxygen.
6. Funciones auxiliares: Se realizará una pequeña descripción de las funciones auxiliares utilizadas, por ejemplo las funciones matemáticas, alfanuméricas, etc.
7. Modo de obtención del intérprete: Se explicarán los directorios y ficheros de cada directorio utilizados por el intérprete para realizar todo el análisis.
8. Ejemplos: Se indicarán los ejemplos y ficheros utilizados para comprobar el buen funcionamiento del analizador.
9. Conclusiones: Se explicaran los puntos fuertes y débiles del intérprete, así como una reflexión final sobre el trabajo realizado.
10. Bibliografía: Se indicarán los enlaces utilizados para la realización del trabajo.
11. Anexos: Se indicaran aquellos anexos considerados oportunos para mejorar la calidad de la documentación.

## Capítulo 2

# LENGUAJE DEL PSEUDOCÓDIGO

### 2.1. COMPONENTES LÉXICOS

Se han diseñado las siguientes palabras reservadas utilizadas en las sentencias:

1. LEER : { "LEER", LEER }
2. LEER\_CADENA: { "LEER\_CADENA", LEER\_CADENA }
3. ESCRIBIR: { "ESCRIBIR", ESCRIBIR }
4. ESCRIBIR\_CADENA: { "ESCRIBIR\_CADENA", ESCRIBIR\_CADENA }
5. SI: { "SI", SI }
6. ENTONCES: { ".<sup>ENTONCES</sup>", ENTONCES }
7. SI\_NO: { "SI\_NO", SI\_NO }
8. FIN\_SI: { "FIN\_SI", FIN\_SI }
9. MIENTRAS: { "MIENTRAS", MIENTRAS }
10. HACER: { "HACER", HACER }

11. FIN\_MIENTRAS: {"FIN\_MIENTRAS", FIN\_MIENTRAS }
12. REPETIR: {REPETIR", REPETIR }
13. DESDE: {REPETIR", REPETIR }
14. HASTA: {"HASTA", HASTA }
15. PASO: {"PASO", PASO }
16. VERDADERO: {"VERDADERO", true }
17. FALSO: {"FALSO", false }
18. #mod: (?i:#mod) { return MODULO;}
19. #div: (?i:#div) { return DIVISION\_ENTERA;}
20. #o: (?i:#o) { return OR; }
21. #y: (?i:#y) { return AND; }
22. #no: (?i:#no) { return NOT; }
23. #borrar: (?i:#borrar) {return BORRAR;}
24. #lugar: (?i:#lugar) {return LUGAR;}

Se han diseñado identificadores con las siguientes características:

1. Estarán compuestos por una serie de letras, dígitos y subrayado
2. Deben comenzar por una letra.
3. No podrán acabar con el símbolo de subrayado, ni tener 2 subrayados seguidos.

Para ello, hemos utilizado las siguientes definiciones regulares:

```
DIGITOS  [0-9]
LETTER  [a-zA-Zá-úÁ-Ú]
SUBRAYADO [\_]
IDENTIFIER  {LETTER}({LETTER}|{DIGITOS}|{SUBRAYADO}({DIGITOS}|{LETTER}))*
```

Se han diseñado números con las siguientes características:



1. Se utilizarán números enteros, reales de punto fijo y reales con notación científica.
2. Todos ellos serán tratados conjuntamente como números

```
NUMBER1 {DIGITOS}+\.?
NUMBER2 {DIGITOS}*\. {DIGITOS}+
NUMBER3 {NUMBER2}[eE][\+|-]?{DIGITOS}*
```

```
{NUMBER1}|{NUMBER2}|{NUMBER3} {
/* Conversion of type and sending of the numerical value to the parser */
yyval.number = atof(yytext);
return NUMBER;
}
```

Se han diseñado cadenas con las siguientes características:

1. Están compuestas por una serie de caracteres delimitados por comillas simples.
2. Deberá permitir la inclusión de la comilla simple utilizando la barra.
3. Las comillas exteriores no se almacenarán como parte de la cadena.

```
CADENA "'\"([^\"]|\"\\\"')*\"'
```

```
{CADENA} {
int longitud = 9999;
char cadena[longitud];
/* Eliminamos la comilla final */
yytext[yytext[0]] = '\0';
for(int i=0; i<= yytext[0]; i++){
/* Eliminamos la comilla inicial */
cadena[i] = yytext[i+1];
}
yyval.identifier = cadena;
return CADENA;
}
```

Se ha implementado el siguiente operador de asignación:

```
" :="      { return ASSIGNMENT; }
```

Se han implementado los siguientes operadores aritméticos:

```
"+"          { return SUMA; }  
"- "         { return RESTA; }  
"* "         { return MULTIPLICACION; }  
"/ "         { return DIVISION; }  
(?i:#div)    { return DIVISION_ENTERA; }  
(?i:#mod)    { return MODULO; }  
" * * "      { return POWER; }
```

Se ha implementado el siguiente operador alfanumérico:

```
" || "      { return CONCATENACION; }
```

Se han implementado los siguientes operadores relacionales:

```
"<"         { return MENOR_QUE; }  
"<="        { return MENOR_O_IGUAL; }  
">"         { return MAYOR_QUE; }  
">="        { return MAYOR_O_IGUAL; }  
"="         { return IGUAL; }  
"<>"        { return NO_IGUAL; }
```

Se han implementado los siguientes operadores lógicos:

```
(?i:#o)      { return OR; }
(?i:#y)      { return AND; }
(?i:#no)     { return NOT; }
```

Se han implementado los siguientes comentarios:

1. De varias líneas: Comentarios delimitados por « y »
2. De una línea: Todo lo que siga al símbolo @ hasta el final de la línea ( pulsar el botón enter)

```
@.*          {;}

"<<"        {BEGIN(COMENTARIOVARIO);}
<COMENTARIOVARIO>[^'\\" { }
<COMENTARIOVARIO>">>" {BEGIN(INITIAL);}
```

Se ha implementado el símbolo ';' para indicar el fin de una sentencia.

```
";" {return SEMICOLON;}
```

## 2.2. SENTENCIAS

Se han implementado las siguientes sentencias de asignación:

1. `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
2. `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.

Para ello, hemos utilizado el código del ejemplo17, añadiendo la expresión de cadena y el operador alfanumérico de concatenación ( se ha añadido al bloque exp del fichero interpreter.y):

```
| CADENA
{
    //Create a new cadena node
    $$ = new lp::StringNode($1);
}
```

```
| CADENA
{
    //Create a new cadena node
    $$ = new lp::StringNode($1);
}
```

Se han implementado las siguientes sentencias del lectura:

1. Leer (identificador): Declara a identificador como variable numérica y le asigna el número leído.
2. Leer\_cadena (identificador): Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).

```

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

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

| LEER_CADENA LPAREN VARIABLE RPAREN
{
    // Create a new read node
    $$ = new lp::ReadChainStmt($3);
}

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

```

Se han implementado las siguientes sentencias de escritura:

1. Escribir (expresion numerica): El valor de la expresión numérica es escrito en la pantalla.
2. Escribir\_cadena (expresion alfanumerica): La cadena sin comillas exteriores es escrita en la pantalla, permitiendo la interpretación de comandos de saltos de línea y tabuladores.

```

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

| ESCRIBIR_CADENA LPAREN exp RPAREN
{
    // Create a new print node
    $$ = new lp::PrintChainStmt($3);
}
;

```

Se han implementado las siguientes sentencias de control:

```

si: /* Simple conditional statement */
   SI controlSymbol cond ENTONCES stmtlist FIN_SI
   {
       // Create a new if statement node
       $$ = new lp::IfStmt($3, $5);

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

   /* Compound conditional statement */
   | SI controlSymbol cond ENTONCES stmtlist SI_NO stmtlist FIN_SI
   {
       // Create a new if statement node
       $$ = new lp::IfStmt($3, $5, $7);

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

```

```

mientras: MIENTRAS controlSymbol cond HACER stmtlist FIN_MIENTRAS
   {
       // Create a new while statement node
       $$ = new lp::WhileStmt($3, $5);

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

```

```

repetir: REPETIR controlSymbol stmtlist HASTA cond
   {
       $$=new lp::RepeatStmt($3, $5);

       control --;
   }
;

```

```

para: PARA controlSymbol VARIABLE DESDE exp HASTA exp HACER stmtlist FIN_PARA
{
    // Create a new for statement node
    $$ = new lp::ParaStmt($3,$5,$7,$9);
    // Control del modo interactivo
    control --;
}

| PARA controlSymbol VARIABLE DESDE exp HASTA exp PASO exp HACER stmtlist FIN_PARA
{
    // Create a new for statement node
    $$ = new lp::ParaStmt($3, $5, $7, $9, $11);

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

;

```

Se han implementado los siguientes comandos especiales:

1. #borrar: Borra la pantalla.
2. #lugar(expresion numerica1, expresion numerica2): Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

```

borrar:
{
    BORRAR
    {
        //Create a new borrar node
        $$ = new lp::BorrarStmt();
    }
}

;

```

```

lugar: LUGAR LPAREN exp COMMA exp RPAREN
{
    // Create a new lugar node
    $$ = new lp::LugarStmt($3,$5);
}

;

```

## Capítulo 3

# TABLA DE SÍMBOLOS

En este apartado se describirán las clases utilizadas en la practica en lo referente a la tabla de símbolos:

La clase `SymbolInterface` es la clase prototipo de los métodos virtuales puros. Esta clase no tiene parámetros. Esta clase cuenta con las funciones miembro `getName`, `getToken`, `setName` y `setToken`. De esta clase deriva la clase `Symbol`.

La clase `Symbol` hereda el public de la clase `SymbolInterface`. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto cuyos parámetros son dos variables, una variable `name` que es el nombre de lo `Symbol` y la variable `token` que es el token de `Symbol`, y otro constructor que es una copia cuyos parámetros son una variable `s` que es un objeto de la clase `Symbol`. En el primer constructor se crea un nuevo `Symbol` con los valores de los parámetros y en el segundo se crea con los valores de un `Symbol` existente. Esta clase cuenta con las funciones miembro `getName`, `getToken`, `operator<`, `operator==`, `setName` y `setToken`. De esta clase derivan:

1. La clase `Builtin` hereda el public de la clase `Symbol`. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de `Symbol` como inicializador de miembros, cuyos parámetros son tres variables, una variable `name` que es el nombre de `Builtin`, la variable `token` que es el token de `Builtin` y la variable `nParameters` que son los `n` parámetros de `Builtin`, y otro constructor que es una copia cuyos parámetros son una variable



s que es un objeto de la clase Symbol. En el primer constructor se crea un nuevo Builtin con los valores de los parámetros y en el segundo se crea con los valores de un Builtin existente. Esta clase cuenta con las funciones miembro getNParameters, operator=, read, setNParameters y write. De esta clase derivan:

- La clase BuiltinParameter0 hereda el public de la clase Builtin. Esta clase tiene dos constructores, un constructor que utiliza una función inline que usa el constructor de Builtin como inicializador de miembros, cuyos parámetros son cuatro variables, una variable name que es el nombre de la BuiltinParameter0, la variable token que es el token de BuiltinParameter0, la variable nParameters que es el número de parámetros de BuiltinParameter0 y la variable function que es una función numérica del BuiltinParameter0, y otro constructor que es una copia cuyos parámetros son una variable f que es un objeto de la clase BuiltinParameter0. En el primer constructor se crea un nuevo BuiltinParameter0 con las funciones de los parámetros y en el segundo se crea un nuevo BuiltinParameter0 con las funciones de un BuiltinParameter0 ya existente. Esta clase cuenta con las funciones miembro getFunction, operator= y setFunction.

- La clase BuiltinParameter1 hereda el public de la clase Builtin. Esta clase tiene dos constructores, un constructor que utiliza una función inline que usa el constructor de Builtin como inicializador de miembros, cuyos parámetros son cuatro variables, una variable name que es el nombre de la BuiltinParameter1, la variable token que es el token de BuiltinParameter1, la variable nParameters que es el número de parámetros de BuiltinParameter1 y una variable function que es una función numérica del BuiltinParameter1, y otro constructor que es una copia cuyos parámetros son una variable f que es un objeto de la clase BuiltinParameter1. En el primer constructor se crea un nuevo BuiltinParameter1 con las funciones de los parámetros y en el segundo se crea un nuevo BuiltinParameter1 con las funciones de un BuiltinParameter1 ya existente. Esta clase cuenta con las funciones miembro getFunction, operator= y setFunction.

- La clase BuiltinParameter2 hereda el public de la clase Builtin. Esta clase tiene dos constructores, un constructor que utiliza una función inline que usa el constructor de Builtin como inicializador de miembros, cuyos parámetros son cuatro variables,

una variable `name` que es el nombre de la `BuiltinParameter2`, la variable `token` que es el token de `BuiltinParameter2`, la variable `nParameters` que es el número de parámetros de `BuiltinParameter2` y una variable `function` que es una función numérica del `BuiltinParameter2`, y otro constructor que es una copia cuyos parámetros son una variable `f` que es un objeto de la clase `BuiltinParameter2`. En el primer constructor se crea un nuevo `BuiltinParameter2` con las funciones de los parámetros y en el segundo se crea un nuevo `BuiltinParameter2` con las funciones de un `BuiltinParameter2` ya existente. Esta clase cuenta con las funciones miembro `getFunction`, `operator=` y `setFunction`.

2. La clase `Constant` hereda el `public` de la clase `Symbol`. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de `Symbol` como inicializador de miembros, cuyos parámetros son tres variables, una variable `name` que es el nombre de `Constant`, la variable `token` que es el token de `Constant` y la variable `type` que indica el tipo de `Constant`, y otro constructor que es una copia cuyos parámetros son una variable `c` que es un objeto de la clase `Constant`. En el primer constructor se crea un nuevo `Constant` con los valores de los parámetros y en el segundo se crea un nuevo `Constant` con los valores de un `Constant` existente. Esta clase cuenta con las funciones miembro `getType`, `operator=`, `read`, `setType` y `write`. De esta clase derivan:

- La clase `LogicalConstant` hereda el `public` de la clase `Constant`. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de `Constant` como inicializador de miembros, cuyos parámetros son cuatro variables, una variable `name` que es el nombre de `LogicalConstant`, la variable `token` que es el token de `LogicalConstant`, la variable `type` que es el tipo de `LogicalConstant` y una variable `value` que es un valor lógico del `LogicalConstant`, y otro constructor que es una copia cuyos parámetros son una variable `n` que es un objeto de la clase `LogicalConstant`. En el primer constructor se crea un nuevo `LogicalConstant` con los valores de los parámetros y en el segundo se crea un nuevo `LogicalConstant` con los valores de un `LogicalConstant` ya existente. Esta clase cuenta con las funciones miembro `getValue`, `operator=`, `read`, `setValue` y `write`. Esta clase también tiene las funciones `operator«` y `operator»` como funciones

Friends.

- La clase NumericConstant hereda el public de la clase Constant. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de Constant como inicializador de miembros, cuyos parámetros son cuatro variables, una variable name que es el nombre de NumericConstant, la variable token que es el token de NumericConstant, la variable type que es el tipo de NumericConstant y una variable value que es un valor numérico del NumericConstant, y otro constructor que es una copia cuyos parámetros son una variable n que es un objeto de la clase NumericConstant. En el primer constructor se crea un nuevo NumericConstant con los valores de los parámetros y en el segundo se crea un nuevo NumericConstant con los valores de un NumericConstant ya existente. Esta clase cuenta con las funciones miembro getValue, operator=, read, setValue y write. Esta clase también tiene las funciones operator« y operator» como funciones Friends.
- 3. La clase Keyword hereda el public de la clase Symbol. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de Symbol como inicializador de miembros, cuyos parámetros son dos variables, una variable name que es el nombre de Keyword y la variable token que es el token de Keyword, y otro constructor que es una copia cuyos parámetros son una variable k que es un objeto de la clase Keyword. En el primer constructor se crea un nuevo Keyword con los valores de los parámetros y en el segundo se crea un nuevo Keyword con los valores de un Keyword existente. Esta clase cuenta con las funciones miembro operator=, read y write.
- 4. La clase Variable hereda el public de la clase Symbol. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de Symbol como inicializador de miembros, cuyos parámetros son tres variables, una variable name que es el nombre de la Variable, la variable token que es el token de Variable y la variable type que es el tipo de la Variable, y otro constructor que es una copia cuyos parámetros son una variable s que es un objeto de la clase Variable. En el primer constructor se crea un nuevo Variable con los valores de los parámetros y en el segundo se crea un nuevo Variable con los

valores de un Variable ya existente existente. Esta clase cuenta con las funciones miembro getType, operator=, read, setType y write. De esta clase derivan:

- La clase LogicalVariable hereda el public de la clase Variable. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de Variable como inicializador de miembros, cuyos parámetros son cuatro variables, una variable name que es el nombre de LogicalVariable, la variable token que es el token de LogicalVariable, la variable type que es el tipo de LogicalVariable y una variable value que es un valor numérico de LogicalVariable, y otro constructor que es una copia cuyos parámetros son una variable n que es un objeto de la clase LogicalVariable. En el primer constructor se crea un nuevo LogicalVariable con los valores de los parámetros y en el segundo se crea un nuevo LogicalVariable con los valores de un LogicalVariable ya existente. Esta clase cuenta con las funciones miembro getValue, operator=, read, setValue y write. Esta clase también tiene las funciones operator« y operator» como funciones Friends.

- La clase NumericVariable hereda el public de la clase Variable. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de Variable como inicializador de miembros, cuyos parámetros son cuatro variables, una variable name que es el nombre de NumericVariable, la variable token que es el token de NumericVariable, la variable type que es el tipo de NumericVariable y una variable value que es un valor numérico de NumericVariable, y otro constructor que es una copia cuyos parámetros son una variable n que es un objeto de la clase NumericVariable. En el primer constructor se crea un nuevo NumericVariable con los valores de los parámetros y en el segundo se crea un nuevo NumericVariable con los valores de un NumericVariable ya existente. Esta clase cuenta con las funciones miembro getValue, operator=, read, setValue y write. Esta clase también tiene las funciones operator« y operator» como funciones Friends.

- La clase StringVariable hereda el public de la clase Variable. Esta clase tiene dos constructores, un constructor con argumentos con valores por defecto que utiliza una función inline que usa el constructor de Variable como inicializador de miem-

bros, cuyos parámetros son cuatro variables, una variable name que es el nombre de StringVariable, la variable token que es el token de StringVariable, la variable type que es el tipo de StringVariable y una variable value que es una cadena de la StringVariable, y otro constructor que es una copia cuyos parámetros son una variable n que es un objeto de la clase StringVariable. En el primer constructor se crea un nuevo StringVariable con los valores de los parámetros y en el segundo se crea un nuevo StringVariable con los valores de un StringVariable ya existente. Esta clase cuenta con las funciones miembro getValue, operator=, read, setValue y write. Esta clase también tiene las funciones operator« y operator» como funciones Friends.

Diagrama de la Tabla de Símbolos en el anexo.

## Capítulo 4

# ANÁLISIS LÉXICO

El análisis léxico ha sido realizado usando las siguientes definiciones regulares:

1. DIGITOS: Reconoce los dígitos que conforman cualquier numero.
2. LETTER: Reconoce todas las letras del abecedario, incluyendo letras mayúsculas, letras minúsculas, vocales acentuadas y vocales sin acentuar.
3. SUBRAYADO: Reconoce el subrayado
4. NUMBER1: Reconoce los números enteros.
5. NUMBER2: Reconoce los números reales.
6. NUMBER3: Reconoce los números con notación científica.
7. CADENA: Reconoce las cadenas.

Además, se han implementado las siguientes expresiones regulares:



Al detectar el analizador léxico un espacio o una tabulación, el analizador léxico lo sal-

ta.

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

Al detectar el analizador léxico un salto de línea, el analizador léxico aumenta en 1 el número de líneas que ha analizado. Esto es necesario para el control de errores, pues permite detectar la línea en la cuál se ha producido un error durante el análisis sintáctico.

```
";" {return SEMICOLON;}
```

Al detectar el analizador léxico el símbolo ';', el analizador léxico le indica al analizador sintáctico que se ha detectado el símbolo SEMICOLON.

```
", " {return COMMA;}
```

Al detectar el analizador léxico el símbolo ',', el analizador léxico le indica al analizador sintáctico que se ha detectado un el símbolo COMMA.



```
{CADENA} {
int longitud = 9999;
char cadena[longitud];
/* Eliminamos la comilla final */
yytext[yytext[0]] = '\0';
for(int i=0; i<= yytext[0]; i++){
    /* Eliminamos la comilla inicial */
    cadena[i] = yytext[i+1];
}
yyval.identifier = cadena;
return CADENA;
}
```

Al detectar el analizador léxico una cadena, elimina de dicha cadena las comillas exteriores y le indica al analizador sintáctico que ha detectado una cadena ( sin las comillas

exteriores).

```
{NUMBER1}|{NUMBER2}|{NUMBER3} {  
    /* Conversion of type and sending of the numerical value to the parser */  
    yyval.number = atof(yytext);  
    return NUMBER;  
}
```

Al detectar el analizador léxico un número, le indica al analizador sintáctico el número detectado.

```
{IDENTIFIER}   
    /* NO SE DISTINGUEN ENTRE MAYUSCULAS Y MINUSCULAS */  
    for (int i = 0; i<=yytext[i]; i++){yytext[i] = toupper(yytext[i]);}  
    std::string identifier(yytext);  
    /* strdup() function returns a pointer to a new string which is a duplicate of the string yytext*/  
    yyval.identifier = strdup(yytext);  
    /* If the identifier is not in the table of symbols then it is inserted */  
    if (table.lookupSymbol(identifier) == false)  
    {  
        /*  
         * The identifier is inserted into the symbol table as undefined Variable with value 0.0  
         */  
        lp::NumericVariable *n = new lp::NumericVariable(identifier,VARIABLE,UNDEFINED,0.0);  
        /*  
         * A pointer to the new NumericVariable is inserted into the table of symbols  
         */  
        table.installSymbol(n);  
        return VARIABLE;  
    }  
    /* If the identifier is in the table then its token is returned */  
    else  
    {  
        lp::Symbol *s = table.getSymbol(identifier);  
        return s->getToken();  
    }  

```

Al detectar el analizador léxico un identificador ( no se distingue entre mayúsculas ni minúsculas), el analizador léxico comprueba si el identificador no está guardado en la tabla de símbolos ( en cuyo caso guarda el identificador en la tabla de símbolos) o si el identificador está guardado en la tabla de símbolos, en cuyo caso obtiene el componente léxico del identificador y se lo notifica al analizador sintáctico.

```
"+" { return SUMA;}
```

Al detectar el analizador léxico el símbolo '+', le notifica al analizador léxico que ha encontrado el símbolo SUMA.



```
"_ " { return RESTA;}
```

Al detectar el analizador léxico el símbolo '-', le notifica al analizador léxico que ha encontrado el símbolo RESTA.

```
"*" { return MULTIPLICACION;}
```

Al detectar el analizador léxico el símbolo '\*', le notifica al analizador léxico que ha encontrado el símbolo MULTIPLICACION.

```
"/" {return DIVISION; }
```

Al detectar el analizador léxico el símbolo '/', le notifica al analizador léxico que ha encontrado el símbolo DIVISION.

```
(?i:#div) { return DIVISION_ENTERA;}
```

Al detectar el analizador léxico el símbolo '#div' ( sin distinción de mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo DIVISION\_ENTERA.

```
(?i:#mod) { return MODULO;}
```

Al detectar el analizador léxico el símbolo '#mod' ( sin distinción de mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo MODULO.

```
"**" { return POWER;}
```

Al detectar el analizador léxico el símbolo '\*\*', le notifica al analizador léxico que ha encontrado el símbolo POWER.

```

"||"      { return CONCATENACION; }
"<"      { return MENOR_QUE; }
"<="     { return MENOR_O_IGUAL; }
">"      { return MAYOR_QUE; }
">="     { return MAYOR_O_IGUAL; }
"="       { return IGUAL; }
"<>"     { return NO_IGUAL; }
(?i:#o)   { return OR; }
(?i:#y)   { return AND; }
(?i:#no)  { return NOT; }
@.*       {;}
"<<"     {BEGIN(COMENTARIOVARIO);}
<COMENTARIOVARIO>[^'\\" ] {}
<COMENTARIOVARIO>">>" {BEGIN(INITIAL);}

"("       { return LPAREN; }

")"       { return RPAREN; }

":="      { return ASSIGNMENT;}

(?i:#borrar) {return BORRAR;}

(?i:#lugar) {return LUGAR;}

```

Al detectar el analizador léxico el símbolo '||', le notifica al analizador léxico que ha encontrado el símbolo CONCATENACION.

Al detectar el analizador léxico el símbolo '<', le notifica al analizador léxico que ha encontrado el símbolo MENOR\_QUE.

Al detectar el analizador léxico el símbolo '<=', le notifica al analizador léxico que ha encontrado el símbolo MENOR\_O\_IGUAL.

Al detectar el analizador léxico el símbolo '>', le notifica al analizador léxico que ha encontrado el símbolo MAYOR\_QUE.

Al detectar el analizador léxico el símbolo '>=', le notifica al analizador léxico que ha encontrado el símbolo MAYOR\_O\_IGUAL.

Al detectar el analizador léxico el símbolo '<>', le notifica al analizador léxico que ha encontrado el símbolo NO\_IGUAL.

Al detectar el analizador léxico el símbolo '#o' (sin distinción entre mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo OR.

Al detectar el analizador léxico el símbolo '#y' (sin distinción entre mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo AND.

Al detectar el analizador léxico el símbolo '#no' (sin distinción entre mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo NOT.

Al detectar el analizador léxico el símbolo '@', el analizador léxico se salta todo lo que encuentre hasta encontrar el salto de línea.

Al detectar el analizador léxico el símbolo '«', el analizador léxico se salta todo lo que encuentre hasta encontrar el símbolo '»'.

Al detectar el analizador léxico el símbolo '(', le notifica al analizador léxico que ha encontrado el símbolo LPAREN.

Al detectar el analizador léxico el símbolo ')', le notifica al analizador léxico que ha encontrado el símbolo RPAREN.

Al detectar el analizador léxico el símbolo ':=', le notifica al analizador léxico que ha encontrado el símbolo ASSIGNMENT.

Al detectar el analizador léxico el símbolo '#borrar' (sin distinción entre mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo BORRAR.

Al detectar el analizador léxico el símbolo '#lugar' (sin distinción entre mayúsculas y minúsculas), le notifica al analizador léxico que ha encontrado el símbolo LUGAR.

```

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

<<EOF>> { /* The interpreter finishes when finds the end of file character */
    |
    |   return 0;
    |
}

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

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

```

Al detectar el analizador léxico el símbolo '{', le notifica al analizador léxico que ha encontrado el símbolo LETFCURLYBRACKET.

Al detectar el analizador léxico el símbolo '}', le notifica al analizador léxico que ha encontrado el símbolo RIGHTCURLYBRACKET.

Al detectar el analizador léxico el símbolo 'EOF', finaliza el interpreter debido a que ha encontrado el final del fichero ( esto solo se aplica cuando se realiza la ejecución del intérprete a partir de un fichero).

Al detectar cualquier símbolo no indicado anteriormente, el analizador léxico lo trataría como un error.

# Capítulo 5

## ANÁLISIS SINTÁCTICO

### 5.1. SÍMBOLOS DE LA GRAMÁTICA

#### 5.1.1. SÍMBOLOS TERMINALES (COMPONENTES LÉXICOS)

Los símbolos terminales utilizados en la practica han sido:

- SEMICOLON ( ; )
- DOUBLEPOINTS ( : )
- LPAREN ( ( )
- RPAREN ( ) )
- COMMA ( , )
- LETFCURLYBRACKET ( { )
- RIGHTCURLYBRACKET ( } )
- OR ( #o )
- AND ( #y )
- NOT ( #no )

- MENOR\_QUE ( < )
- MAYOR\_QUE ( > )
- MENOR\_O\_IGUAL ( <= )
- MAYOR\_O\_IGUAL ( >= )
- IGUAL ( = )
- NO\_IGUAL ( <> )
- POWER ( \*\* )
- ASSIGNMENT ( := )
- SUMA ( + )
- RESTA ( - )
- MULTIPLICACION ( \* )
- DIVISION ( / )
- MODULO ( #mod )
- DIVISION\_ENTERA ( #div )
- CONCATENATION ( || )
- UNARY ( +/- NUMERO )
- NUMBER
- CADENA
- BOOL
- VARIABLE
- UNDEFINED
- CONSTANT

- BULTIN
- BORRAR
- LUGAR
- Símbolos terminales de las sentencias de control: - ESCRIBIR
- ESCRIBIR\_CADENA
- LEER
- LEER\_CADENA
- SI
- SI\_NO
- ENTONCES
- FIN\_SI
- MIENTRAS
- HACER
- FIN\_MIENTRAS
- REPETIR
- HASTA
- PARA
- DESDE
- PASO
- FIN\_PARA

### 5.1.2. SÍMBOLOS NO TERMINALES

Los símbolos no terminales utilizados en la practica han sido:

- exp (Expresión)
- cond (Condición)
- listOfExp (lista de expresiones)
- restOfListExp (resto de lista de expresiones)
- stmtlist (lista de sentencias)
- stmt (Sentencia)
- asgn (Asignación)
- escribir (Escribir número)
- leer (Leer número)
- escribir\_cadena (Escribir cadena)
- leer\_cadena (Leer cadena)
- si (no terminal de sentencia de control)
- mientras (no terminal de sentencia de control)
- block (no terminal de sentencia de control)
- repetir (no terminal de sentencia de control)
- borrar (Limpiar pantalla)
- lugar (Posicionar cursor en pantalla)
- program (Programa)



## 5.2. REGLAS DE PRODUCCIÓN DE LA GRAMÁTICA

Las reglas de producción de la gramática utilizados en la practica han sido:

- program : stmtlist
- stmtlist: epsilon | stmtlist stmt | stmtlist error
- stmt: SEMICOLON | asgn SEMICOLON | escribir SEMICOLON | leer SEMICOLON | leer\_cadena SEMICOLON | lugar SEMICOLON | si SEMICOLON | borrar SEMICOLON | mientras SEMICOLON | block | repetir SEMICOLON | para SEMICOLON
- block: LETFCURLYBRACKET stmtlist RIGHTCURLYBRACKET
- controlSymbol: épsilon
- lugar: LUGAR LPAREN exp COMMA exp RPAREN
- si:SI controlSymbol cond ENTONCES stmtlist FIN\_SI | SI controlSymbol cond ENTONCES stmtlist SI\_NO stmtlist FIN\_SI
- borrar: BORRAR
- repetir: REPETIR controlSymbol stmtlist HASTA cond
- mientras: MIENTRAS controlSymbol cond HACER stmtlist FIN\_MIENTRAS
- para: PARA controlSymbol VARIABLE DESDE exp HASTA exp HACER stmtlist FIN\_PARA | PARA controlSymbol VARIABLE DESDE exp HASTA exp PASO exp HACER stmtlist FIN\_PARA
- cond: LPAREN exp RPAREN
- asgn: VARIABLE ASSIGNMENT exp | VARIABLE ASSIGNMENT asgn | CONSTANT ASSIGNMENT exp | CONSTANT ASSIGNMENT asgn
- escribir: ESCRIBIR exp | ESCRIBIR\_CADENA LPAREN exp RPAREN

- leer: LEER LPAREN VARIABLE RPAREN | LEER LPAREN CONSTANT RPAREN

- leer\_cadena: LEER\_CADENA LPAREN VARIABLE RPAREN | LEER\_CADENA LPAREN CONSTANT RPAREN

- exp: NUMBER | CADENA | exp SUMA exp | exp RESTA exp | exp MULTIPLICACION exp | exp DIVISION exp | exp DIVISION\_ENTERA exp | LPAREN exp RPAREN | SUMA exp %prec UNARY | RESTA exp %prec UNARY | exp MODULO exp | exp POWER exp | VARIABLE | CONSTANT | BUILTIN LPAREN listOfExp RPAREN | exp MAYOR\_QUE exp | exp MAYOR\_O\_IGUAL exp | exp MENOR\_QUE exp | exp MENOR\_O\_IGUAL exp | exp IGUAL exp | exp NO\_IGUAL exp | exp AND exp | exp OR exp | NOT exp | exp CONCATENACION exp

- listOfExp: Lista vacía de expresiones numéricas | exp restOfListOfExp

- restOfListOfExp: Lista vacía de expresiones numéricas | COMMA exp restOfListOfExp

### 5.3. ACCIONES SEMÁNTICAS

- si: SI controlSymbol cond ENTONCES stmtlist FIN\_SI | SI controlSymbol cond ENTONCES stmtlist SI\_NO stmtlist FIN\_SI

Se crea un nodo IfStmt al cual se le pasa la condición y el bloque de ejecución y, en caso de tener también el entonces, también se le pasa el bloque de sentencias del entonces al IfStmt. Cuando se haya evaluado este nodo, se comprobará que la condición que se le ha pasado de como resultado un boolean y de no ser así dará error. Si la condición da verdadero entonces se ejecuta el bloque de código del entonces, si no se da el caso, si existe el bloque del si\_no se ejecuta ese bloque de código.

- mientras: MIENTRAS controlSymbol cond HACER stmtlist FIN\_MIENTRAS

Se crea un nodo WhileStmt al cual se le pasa la condición que se ha de cumplir y el bloque de código del mientras. Se comprueba que la condición sea una expresión que devuelva como resultado un boolean. Entonces se ejecuta el bloque de código del mientras, mientras

la condición sea cierta

- repetir: REPETIR controlSymbol stmtlist HASTA cond

Se crea un nodo RepeatStmt al que se le pasa el bloque de código del repetir y la condición. Revisa que la condición que se le ha pasado es una expresión que devuelve un boolean y si es cierto se repite la ejecución del bucle hasta que deje de cumplirse, con la salvedad de que en ese caso se va a ejecutar como mínimo una vez. En el caso de la condición no sea un boolean enviará un mensaje de error y dejara de ejecutar el bucle.

- para: PARA controlSymbol VARIABLE DESDE exp HASTA exp HACER stmtlist  
FIN\_PARA | PARA controlSymbol VARIABLE DESDE exp HASTA exp PASO exp  
HACER stmtlist FIN\_PARA

En el bucle para hay dos opciones, pasarle un paso o no, si se le pasa un paso entonces se crea un nodo ForStmt al que se le pasa la variable contador, el valor inicial de esta, el valor de la condición de parada y como se modifica la variable. Si el paso no se le indica se creará un nodo con el resto de valores y luego internamente en la ejecución tendrá como valor 1. Cuando se ejecuta el nodo, se comprueban que las expresiones pasadas sean numéricas y en el caso de que alguna no lo fuera, se dejaría de ejecutar el nodo y se enviaría un mensaje de error. En el caso de existir una variable contador y no ser numérica mostrará un mensaje de error y finalizaría la ejecución, si no existiera la crea. También se hacen comprobaciones de que el bucle no sea infinito ya sea por la variable paso sea 0 o porque el paso nunca lleve a la variable contador al estado final. Por último, tras pasar todas las comprobaciones de forma satisfactoria ya realiza el bucle para correctamente.

## Capítulo 6

# CÓDIGO DE AST

En este apartado se describirán las clases que se han utilizado y creado para la practica:

La clase `ExpNode`, esta clase define los atributos y métodos de la clase, también cuenta con las funciones miembro `evaluateBool`, `evaluateNumber`, `evaluateString`, `getType` y `print`. De esta clase derivan:

1. La clase `BuiltinFunctionNode`: esta clase hereda el public de la clase `ExpNode`. Recibe como parámetros una cadena con el nombre de la función incorporada y crea una nueva `BuiltinFunction` con los parámetros.
  - La clase `BuiltinFunctionNode_0`, hereda el public de la clase `BuiltinFunctionNode` y agrega sus propias funciones de impresión y evaluación. El constructor de `BuiltinFunctionNode_0` usa el constructor de `BuiltinFunctionNode` como inicializador de miembros. Tiene como parámetros una cadena con el nombre de la función incorporada. Cuenta con las funciones miembro `evaluateNumber`, `getType` y `print`.
  - La clase `BuiltinFunctionNode_1`, hereda el public de la clase `BuiltinFunctionNode` y agrega sus propias funciones de impresión y evaluación. El constructor de `BuiltinFunctionNode_1` usa el constructor de `BuiltinFunctionNode` como inicializador de miembros. Tiene como parámetros una cadena con el nombre de la función incorporada y un puntero a `ExpNode`, argumento de `BuiltinFunctionNode_1`. Cuenta con las funciones miembro `evaluateNumber`, `getType` y `print`.

- La clase `BuiltinFunctionNode_2`, hereda el public de la clase `BuiltinFunctionNode` y agrega sus propias funciones de impresión y evaluación. El constructor de `BuiltinFunctionNode_2` usa el constructor de `BuiltinFunctionNode` como inicializador de miembros. Tiene como parámetros una cadena con el nombre de la función incorporada, un puntero a `ExpNode`, primer argumento del `BuiltinFunctionNode`, y un puntero a `ExpNode`, segundo argumento del `BuiltinFunctionNode`. Cuenta con las funciones miembro `evaluateNumber`, `getType` y `print`.
2. La clase `ConstantNode`: hereda el public de la clase `ExpNode`. Tiene como parámetro una variable "value" de tipo `double`. Cuenta con las funciones miembro `evaluateBool`, `evaluateNumber`, `evaluateString`, `getType` y `print`.
  3. La clase `NumberNode`: hereda el public de la clase `ExpNode`. Tiene como parámetro una variable "value" de tipo `double`. Cuenta con las funciones miembro `evaluateNumber`, `getType` y `print`.
  4. La clase `StringNode`: hereda el public de la clase `ExpNode`. Tiene como parámetro una variable "value" de tipo cadena. Cuenta con las funciones miembro `evaluateString`, `getType` y `print`.
  5. La clase `OperatorNode`: hereda el public de la clase `ExpNode`. El constructor tiene como parámetro una variable `R` de tipo puntero a `ExpNode` y una variable `L` de tipo puntero a `ExpNode`. De esta clase derivan:
    - La clase `LogicalOperatorNode`, hereda la clase public de la clase `OperatorNode`. El constructor de `LogicalOperatorNode` usa el constructor de `OperatorNode` como inicializador de miembros. El constructor usa el constructor de `OperatorNode` como inicializador de miembros. Cuenta con la función miembro `getType`. De esta clase derivan:
      - .- La clase `AndNode`, hereda el public de la clase `LogicalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `LogicalOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateBool` y `print`.
      - .- La clase `OrNode`, hereda el public de la clase `LogicalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor

de LogicalOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateBool y print.

- La clase NumericOperatorNode, hereda la clase public de la clase OperatorNode. El constructor usa el constructor de OperatorNode como inicializador de miembros. Cuenta con la función miembro getType. De esta clase derivan:

.-La clase DivisionEnteraNode, hereda el public de la clase NumericOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de NumericOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateNumber y print.

.-La clase DivisionNode, hereda el public de la clase NumericOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de NumericOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateNumber y print.

.-La clase MinusNode, hereda el public de la clase NumericOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de NumericOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateNumber y print.

.-La clase ModuloNode, hereda el public de la clase NumericOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de NumericOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateNumber y print.

.-La clase MultiplicationNode, hereda el public de la clase NumericOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de NumericOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateNumber y print.

.-La clase PlusNode, hereda el public de la clase NumericOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de NumericOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateNumber y print.

.-La clase `PowerNode`, hereda el public de la clase `NumericOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `NumericOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateNumber` y `print`.

- La clase `RelationalOperatorNode`, hereda la clase public de la clase `OperatorNode`. El constructor usa el constructor de `OperatorNode` como inicializador de miembros. Cuenta con la función miembro `getType`. De esta clase derivan:

.- La clase `EqualNode`, hereda el public de la clase `RelationalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `RelationalOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateBool` y `print`.

.- La clase `GreaterOrEqualNode`, hereda el public de la clase `RelationalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `RelationalOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateBool` y `print`.

.- La clase `GreaterThanNode`, hereda el public de la clase `RelationalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `RelationalOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateBool` y `print`.

.- La clase `LeesOrEqualNode`, hereda el public de la clase `RelationalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `RelationalOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateBool` y `print`.

.- La clase `LessThanNode`, hereda el public de la clase `RelationalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de `RelationalOperatorNode` como inicializador de miembros. Cuenta con las funciones miembro `evaluateBool` y `print`.

.- La clase `NotEqualNode`, hereda el public de la clase `RelationalOperatorNode` y agrega sus propias funciones de impresión y evaluación. Usa el constructor de

RelationalOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateBool y print.

- La clase StringOperatorNode, hereda la clase public de la clase OperatorNode. El constructor usa el constructor de OperatorNode como inicializador de miembros. Cuenta con la función miembro getType. De esta clase derivan:

.- La clase ConcatenationNode, hereda el public de la clase StringOperatorNode y agrega sus propias funciones de impresión y evaluación. Usa el constructor de StringOperatorNode como inicializador de miembros. Cuenta con las funciones miembro evaluateString y print.

Diagrama de las clases en el Anexo

La clase Statement, esta clase define los atributos y métodos usados de la clase Statement. También cuenta con las funciones miembro evaluate y print. De esta clase derivan:

1. La clase AssignmentStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. Tiene como parámetros una cadena como variable de AssignmentStmt, un puntero a ExpNode y un puntero a AssignmentStmt. Se crea una nueva AssignmentStmt con los parámetros. Cuenta con las funciones miembro evaluate y print.
2. La clase BlockStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. Tiene como parámetros una lista de Statement. Se crea una nueva BlockStmt con los parámetros. Cuenta con las funciones miembro evaluate y print.
3. La clase BorrarStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. No tiene parámetros. Se crea un nuevo BorrarStmt. Cuenta con las funciones miembro evaluate y print.
4. La clase EmptyStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. Tiene dos constructores, Constructor de IfStmt único (sin alternativa) con los parámetros condition (siendo ExpNode la condicion) y statement1 (siendo la declaración del consecuente), y constructor de



IfStmt compuesto (con alternativa) con los parámetros condition (siendo ExpNode la condición), statement1 (siendo declaración del consecuente) y statement2 (siendo declaración de la alternativa). En ambos se crea un nuevo IfStmt con los parámetros según el caso. Cuenta con las funciones miembro evaluate y print.

5. La clase IfStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. No tiene parámetros. Se crea un nuevo BorrarrStmt. Cuenta con las funciones miembro evaluate y print.
6. La clase LugarStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. El constructor no tiene parámetros. Se crea un nuevo LugarStmt. Cuenta con las funciones miembro evaluate y print.
7. La clase ParaStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable condition, siendo ExpNode la condición, y una variable statement, siendo la primera declaración. Se crea un nuevo ParaStmt con los parámetros. Cuenta con las funciones miembro evaluate y print.
8. La clase PrintChainStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable expression que es un puntero a ExpNode. Se crea un nuevo PrintChainStmt con los parámetros. Cuenta con las funciones miembro evaluate y print.
9. La clase PrintStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable expression que es un puntero a ExpNode. Se crea un nuevo PrintStmt con los parámetros. Cuenta con las funciones miembro evaluate y print.
10. La clase ReadChainStmt: esta clase hereda el public de la clase Statement y agrega sus propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable id que es una cadena. Se crea un nuevo ReadChainStmt con los parámetros. Cuenta con las funciones miembro evaluate y print.
11. La clase ReadStmt: esta clase hereda el public de la clase Statement y agrega sus

propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable `id` que es una cadena. Se crea un nuevo `ReadStmt` con los parámetros. Cuenta con las funciones miembro `evaluate` y `print`.

12. La clase `RepeatStmt`: esta clase hereda el `public` de la clase `Statement` y agrega sus propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable `condition`, siendo `ExpNode` la condición, y una variable `statement`, siendo la declaración del cuerpo del bucle. Se crea un nuevo `RepeatStmt` con los parámetros. Cuenta con las funciones miembro `evaluate` y `print`.
13. La clase `WhileStmt`: esta clase hereda el `public` de la clase `Statement` y agrega sus propias funciones de impresión y evaluación. El constructor tiene como parámetros la variable `condition`, siendo `ExpNode` la condición, y una variable `statement`, siendo la declaración del cuerpo del bucle. Se crea un nuevo `WhileStmt` con los parámetros. Cuenta con las funciones miembro `evaluate` y `print`.

Diagrama de las clases en el Anexo

# Capítulo 7

## FUNCIONES AUXILIARES

En este apartado se describirán las funciones auxiliares utilizadas en la practica.

- `double cos(double x)`: esta función devuelve el coseno del ángulo de  $x$  radianes, recibe como parámetros una variable de tipo `double`. Retorna el coseno de la variable expresada en radianes.
- `double sin(double x)`: esta función devuelve el seno del ángulo de  $x$  radianes, recibe como parámetros una variable de tipo `double`. Retorna el seno de la variable expresada en radianes.
- `double atan(double x)`: esta función devuelve el valor principal del arco tangente de la variable, recibe como parámetros una variable de tipo `double`. Retorna el valor del arco tangente de la variable expresada en radianes.
- `std::abs( double x)`: esta función devuelve el valor absoluto del parámetro dado. Recibe como parámetro una variable de tipo `double`. Retorna el valor absoluto de la variable.
- `double ATAN2(double x, double y)`: esta función calcula el  $\text{atan}(x/y)$ , recibe como parámetros dos variables de tipo `double`,  $x$  e  $y$ . Retorna el resultado de  $\text{atan}(x/y)$ .
- `double EXP(double x)`: esta función calcula la exponencial de un número real, recibe como parámetro una variable de tipo `double`. Primero verifica si hay un error en el argumento. Retorna el resultado de  $\exp(x)$ .

- `double INTEGER(double x)`: esta función calcula la parte entera como un número real, recibe como parámetro una variable de tipo `double`. Retorna `(double) (long) x`.
- `double LOG(double x)`: esta función calcula el logaritmo neperiano de un número real, recibe como parámetro una variable de tipo `double`. Primero verifica si hay un error en el argumento. Retorna el resultado de `log(x)`.
- `double LOG10(double x)`: esta función calcula el logaritmo decimal de un número real, recibe como parámetro una variable de tipo `double`. Primero verifica si hay un error en el argumento. Retorna el resultado de `log(x)`.
- `double RANDOM()`: esta función calcula un número aleatorio. No recibe ningún parámetro y retorna el valor de `(double)(long)rand() / RAND_MAX`;
- `double SQRT(double x)`: esta función calcula la raíz cuadrada de un número real, recibe como parámetro una variable de tipo `double`. Primero verifica si hay un error en el argumento. Retorna el resultado de `sqrt(x)`.

## Capítulo 8

# MODO DE OBTENCIÓN DEL INTÉRPRETE

### 8.1. NOMBRE Y DESCRIPCIÓN DE CADA DIRECTORIO

Los directorios utilizados para la obtención del intérprete son los siguientes:

1. Directorio principal: Directorio que contiene todos los directorios, archivos e información para obtener el intérprete.
2. ast: Directorio con la información necesaria para ejecutar la clase AST.
3. error: Directorio con la información de los errores producidos durante el análisis.
4. examples: Directorio con ejemplos para comprobar el funcionamiento del analizador.
5. html: Directorio que almacena toda la documentación del analizador en formato html.
6. includes: Directorio que almacena los macros utilizados por el analizador.
7. parser: Directorio que almacena la información necesaria para el diseño del analizador léxico y el analizador sintáctico.

8. table: Directorio que almacena los archivos de la tabla de símbolos.

## 8.2. NOMBRE Y DESCRIPCIÓN DE CADA FICHERO UTILIZADO DE CADA DIRECTORIO

Ficheros del directorio principal:

1. Control de tareas.txt: Fichero de control de tareas para el trabajo de prácticas.
2. doxyfile: Fichero que almacena las instrucciones para generar el directorio html y todos sus ficheros.
3. interpreter.cpp: Fichero principal del analizador, el cuál inicia el analizador.
4. interpreter.exe: Fichero ejecutable del analizador.
5. makefile: Makefile principal.
6. interpreter.o: Fichero objeto del fichero interpreter.cpp

Ficheros del directorio ast:

1. ast.cpp: Fichero con las funciones de la clase AST.
2. ast.hpp: Fichero con las declaraciones de las funciones de la clase AST.
3. ast.o: Fichero objeto de la clase AST.
4. makefile: Fichero makefile de la clase AST.

Ficheros del directorio error:

1. error.cpp: Fichero con las funciones del controlador de errores.
2. error.hpp: Fichero con las declaraciones de las funciones del controlador de errores.
3. error.o: Fichero objeto del fichero error.cpp
4. makefile: Makefile.

Ficheros del directorio examples:

1. conversion.e: Fichero ejemplo del funcionamiento sin errores del analizador.
2. menu.e: Fichero que comprueba el funcionamiento del analizador.
3. test.e: Fichero que comprueba el funcionamiento del analizador.
4. test-error.txt: Fichero que comprueba si el analizador detecta y notifica errores en el análisis.

Fichero del directorio includes:

1. macros.hpp: Fichero con los macros de la ventana.

Ficheros del directorio parser:

1. interpreter.l: Fichero del analizador lexico.
2. interpreter.y: Fichero del analizador sintáctico.

Ficheros del directorio table:

1. builtin.cpp: Fichero con las funciones de la clase Builtin.
2. builtin.hpp: fichero con las declaraciones de las funciones de la clase Builtin.
3. builtinParameter0.cpp: fichero con las funciones de la clase builtinParameter0
4. builtinParameter0.hpp: fichero con las declaraciones de las funciones de la clase builtinParameter0.
5. builtinParameter1.cpp: fichero con las funciones de la clase builtinParameter1
6. builtinParameter1.hpp: fichero con las declaraciones de las funciones de la clase builtinParameter1.
7. builtinParameter2.cpp: fichero con las funciones de la clase builtinParameter2
8. builtinParameter2.hpp: fichero con las declaraciones de las funciones de la clase builtinParameter2.

9. `constant.cpp`: fichero con las funciones de la clase `Constant`.
10. `constant.hpp`: fichero con las declaraciones de la clase `constant`.
11. `init.cpp`: fichero con las funciones para inicializar la tabla de símbolos
12. `init.hpp`: Fichero con las declaraciones de las funciones para inicializar la tabla de símbolos.
13. `keyword.cpp`: fichero con las funciones de la clase `Keyword`
14. `keyword.hpp`: fichero con las declaraciones de la clase `keyword`.
15. `logicalConstant.cpp`: fichero con las funciones de la clase `logicalConstant`
16. `logicalConstant.hpp`: fichero con las declaraciones de la clase `logicalConstant`.
17. `logicalVariable.cpp`: fichero con las funciones de la clase `logicalVariable`
18. `logicalVariable.hpp`: fichero con las declaraciones de la clase `logicalVariable`.
19. `mathFunction.cpp`: fichero con las funciones de la clase `mathFunction`
20. `mathFunction.hpp`: fichero con las declaraciones de la clase `mathFunction`.
21. `numericConstant.cpp`: fichero con las funciones de la clase `numericConstant`
22. `numericConstant.hpp`: fichero con las declaraciones de la clase `numericConstant`.
23. `numericVariable.cpp`: fichero con las funciones de la clase `numericVariable`
24. `numericVariable.hpp`: fichero con las declaraciones de la clase `numericVariable`.
25. `StringVariable.cpp`: fichero con las funciones de la clase `StringVariable`
26. `StringVariable.hpp`: fichero con las declaraciones de la clase `StringVariable`.
27. `symbol.cpp`: fichero con las funciones de la clase `symbol`
28. `symbol.hpp`: fichero con las declaraciones de la clase `symbol`.
29. `symbolInterface.hpp`: fichero con las declaraciones de la clase `symbolInterface`.

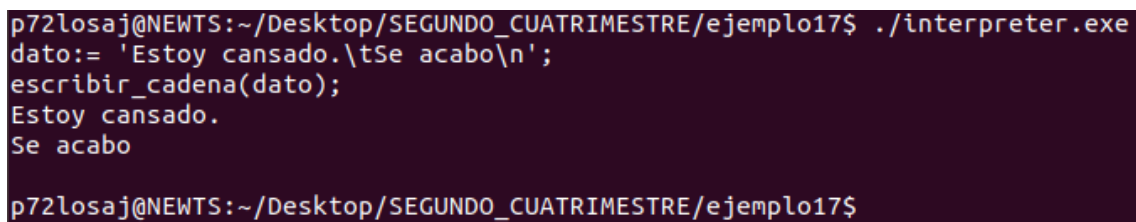


- 30. table.cpp: fichero con las funciones de la clase table
- 31. table.hpp: fichero con las declaraciones de la clase table.
- 32. tableInterface.hpp: fichero con las declaraciones de la clase tableInterface.
- 33. variable.cpp: fichero con las funciones de la clase variable.
- 34. variable.hpp: fichero con las declaraciones de la clase variable.

## Capítulo 9

# MODO DE EJECUCIÓN DEL INTÉRPRETE

### 9.1. INTERACTIVA

A terminal window with a dark purple background. The prompt is 'p72losaj@NEWS:~/Desktop/SEGUNDO\_CUATRIMESTRE/ejemplo17\$'. The command './interpreter.exe' is entered. The script contains two lines: 'dato:= 'Estoy cansado.\tSe acabo\n';' and 'escribir\_cadena(dato);'. The output shows 'Estoy cansado.' followed by a tab character and 'Se acabo' on the next line.

```
p72losaj@NEWS:~/Desktop/SEGUNDO_CUATRIMESTRE/ejemplo17$ ./interpreter.exe
dato:= 'Estoy cansado.\tSe acabo\n';
escribir_cadena(dato);
Estoy cansado.
Se acabo

p72losaj@NEWS:~/Desktop/SEGUNDO_CUATRIMESTRE/ejemplo17$
```

### 9.2. A PARTIR DE UN FICHERO

Para la ejecución del intérprete en modo fichero, se comprobará si la extensión del fichero es correcta ( en caso de que el fichero no sea una extensión .e se mostrará un mensaje de error). Además, se comprobará si el fichero existe ( en caso de que no existiera el fichero, se mostrará un mensaje de error).

```
p72losaj@NEWTs:~/Desktop/SEGUNDO_CUATRIMESTRE/ejemplo17$ ./interpreter.exe examples/test.e
```

MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

Elige una opcion:

Introduce un valor numerico -->

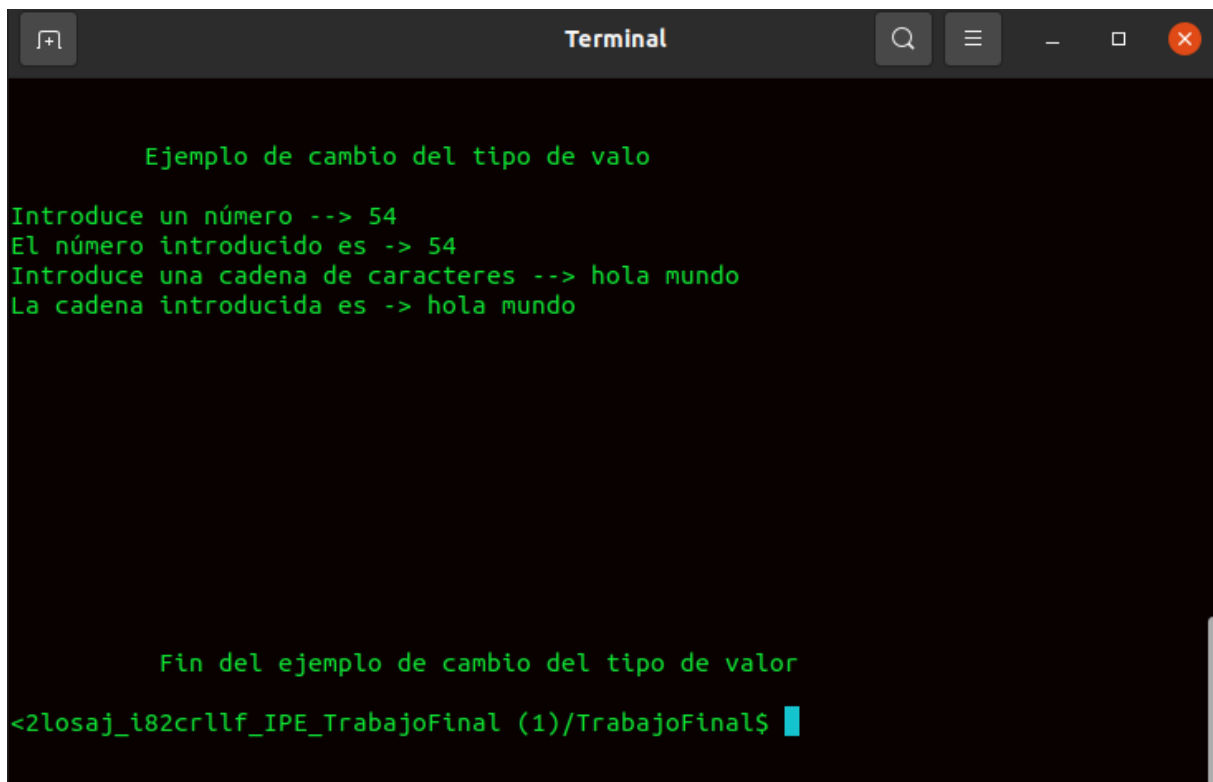
# Capítulo 10

## EJEMPLOS

Se han implementado una serie de ejemplos, los cuales se corresponden con los ficheros del directorio examples.

Estos ficheros son:

- conversion.e: este fichero cambia el tipo de valor introducido por teclado de un número a cadena de caracteres.



```
Terminal

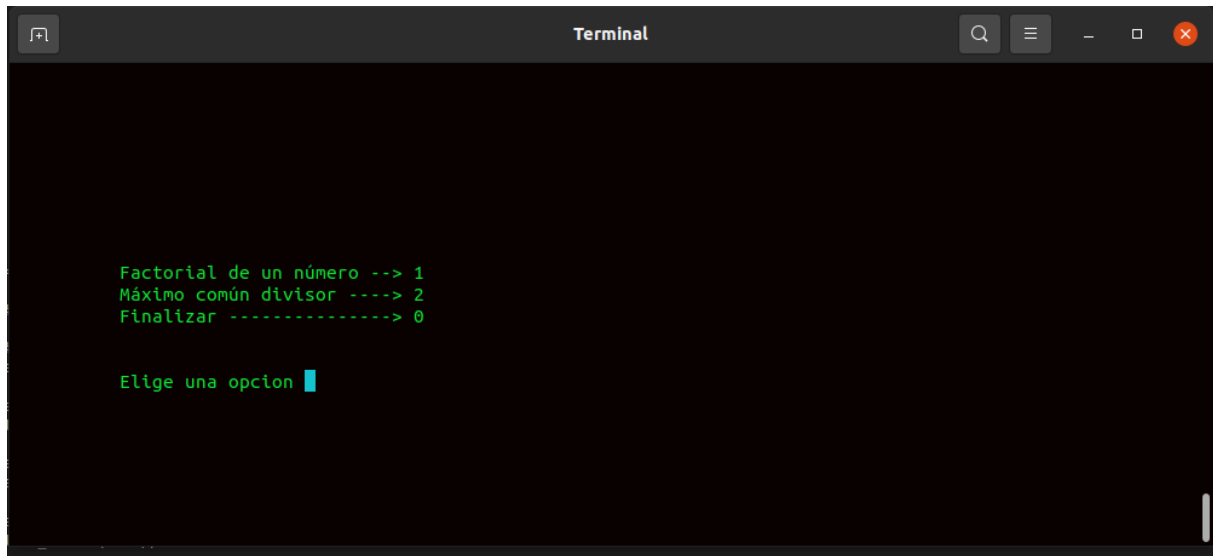
Ejemplo de cambio del tipo de valo

Introduce un número --> 54
El número introducido es -> 54
Introduce una cadena de caracteres --> hola mundo
La cadena introducida es -> hola mundo

Fin del ejemplo de cambio del tipo de valor

<2losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$
```

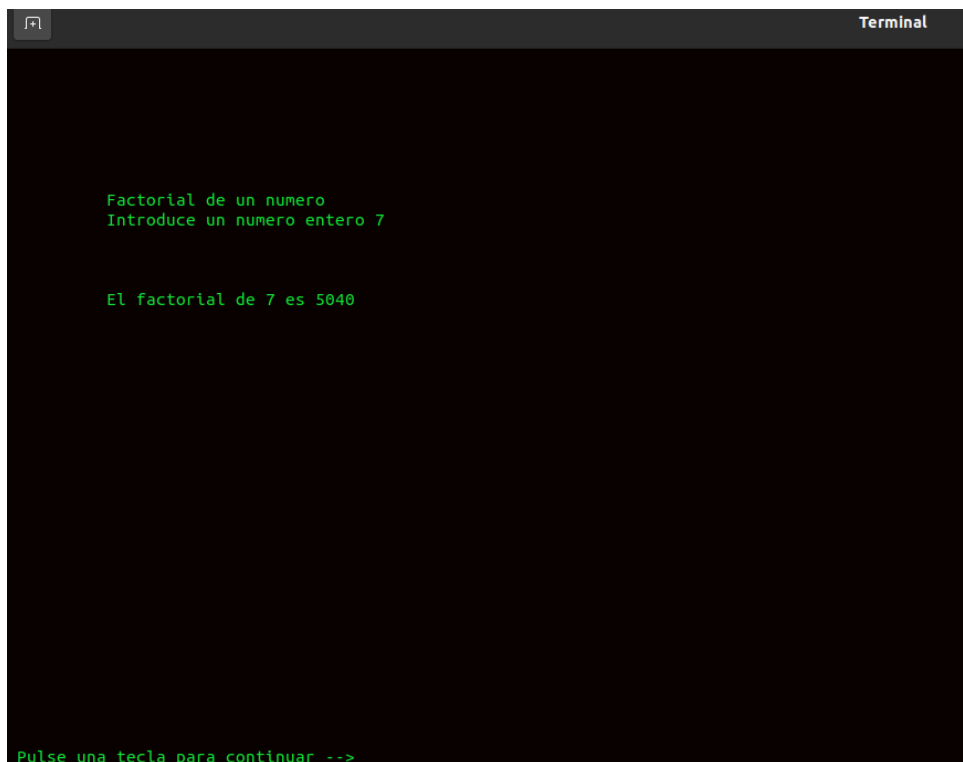
- menu.e: este fichero muestra un pequeño menú en el que se podrá seleccionar si se quiere realizar el factorial de un número o el máximo común divisor.



```
Terminal

Factorial de un número --> 1
Máximo común divisor ----> 2
Finalizar -----> 0

Elige una opción █
```

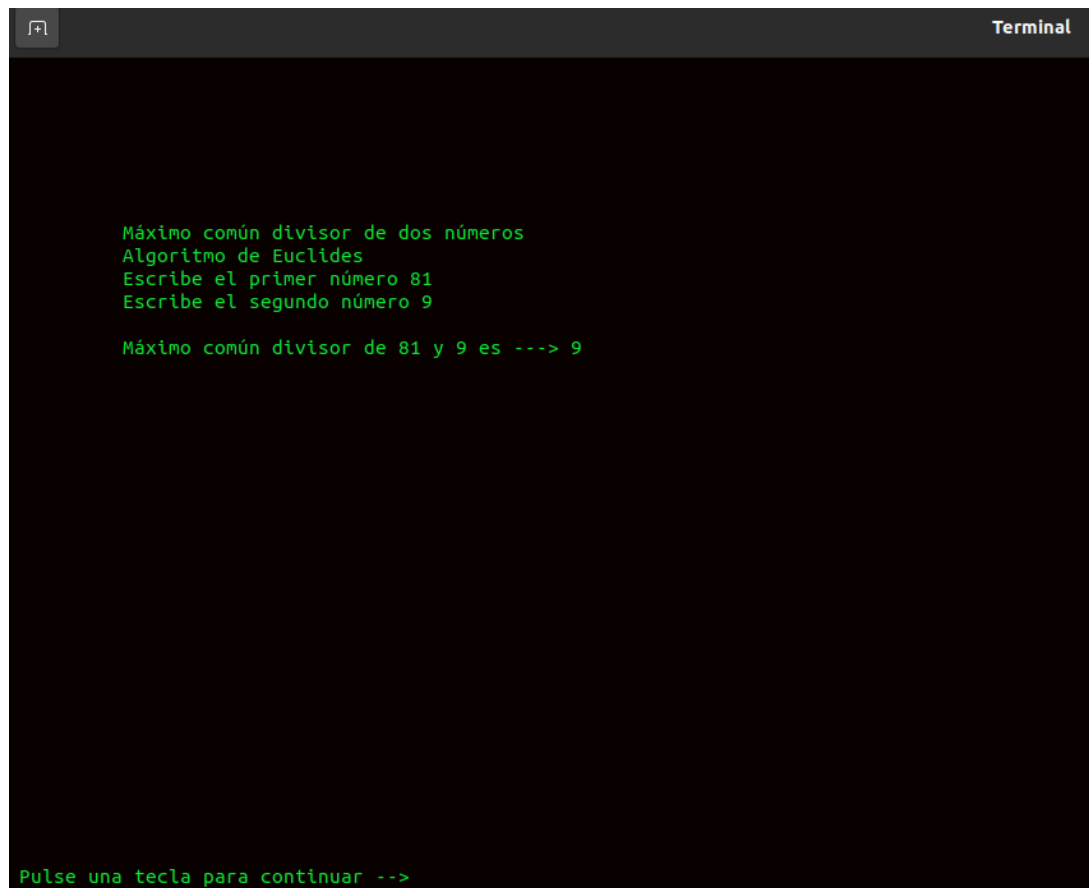


```
Terminal

Factorial de un numero
Introduce un numero entero 7

El factorial de 7 es 5040

Pulse una tecla para continuar -->
```

A terminal window with a dark background and green text. The title bar at the top right says "Terminal". The text inside the terminal reads: "Máximo común divisor de dos números", "Algoritmo de Euclides", "Escribe el primer número 81", "Escribe el segundo número 9", "Máximo común divisor de 81 y 9 es ---> 9", and at the bottom "Pulse una tecla para continuar -->".

```
Terminal

Máximo común divisor de dos números
Algoritmo de Euclides
Escribe el primer número 81
Escribe el segundo número 9

Máximo común divisor de 81 y 9 es ---> 9

Pulse una tecla para continuar -->
```

-test.e: este fichero simula el funcionamiento de una calculadora. Contiene un menú en el que se puede seleccionar la operación que se desee realizar:

1. Suma

```
Terminal
MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

Elige una opcion:
1
Introduce el valor1: 4
Introduce el valor2: 5
Introduce el numero de iteraciones: 1
Resultado de la operacion suma es: 9
```

## 2. Resta

```
Terminal
MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

Elige una opcion:
2
Introduce el valor1: 9
Introduce el valor2: 2
Introduce el numero de iteraciones deseadas: 2
Resultado de la operacion resta es: 5
```

### 3. Multiplicación

```
MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

Elige una opcion:
3
Introduce el valor1: 6
Introduce el valor2: 4
Introduce el numero de iteraciones deseadas: 1
Resultado de la operacion multiplicacion es: 24
```

### 4. División

```
Terminal
MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

Elige una opcion:
4
Introduce el valor1: 90
Introduce el valor2: 5
Introduce el numero de iteraciones deseadas: 1
Resultado de la operacion division es: 18
```



## 5. Potencia

```
Terminal
MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

Elige una opcion:
5
Introduce el valor1: 3
Introduce el valor2: 2
Introduce el numero de iteraciones deseadas: 1
Resultado de la operacion potencia es: 9
```

## 6. Módulo

```
Terminal
MENU

0. Salir

1. Operacion de suma

2. Operacion de resta

3. Operacion de multiplicacion

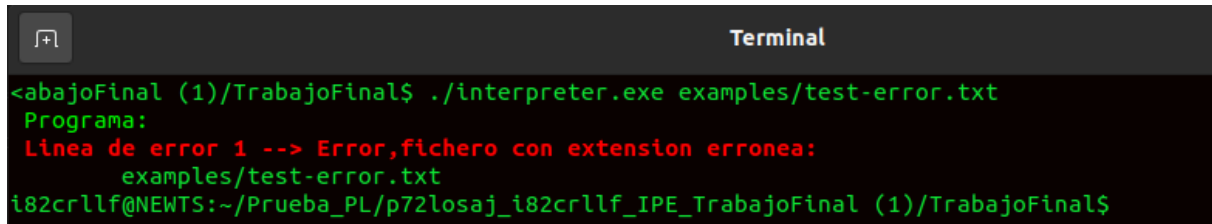
4. Operacion de division

5. Operacion de potencia

6. Operacion de modulo

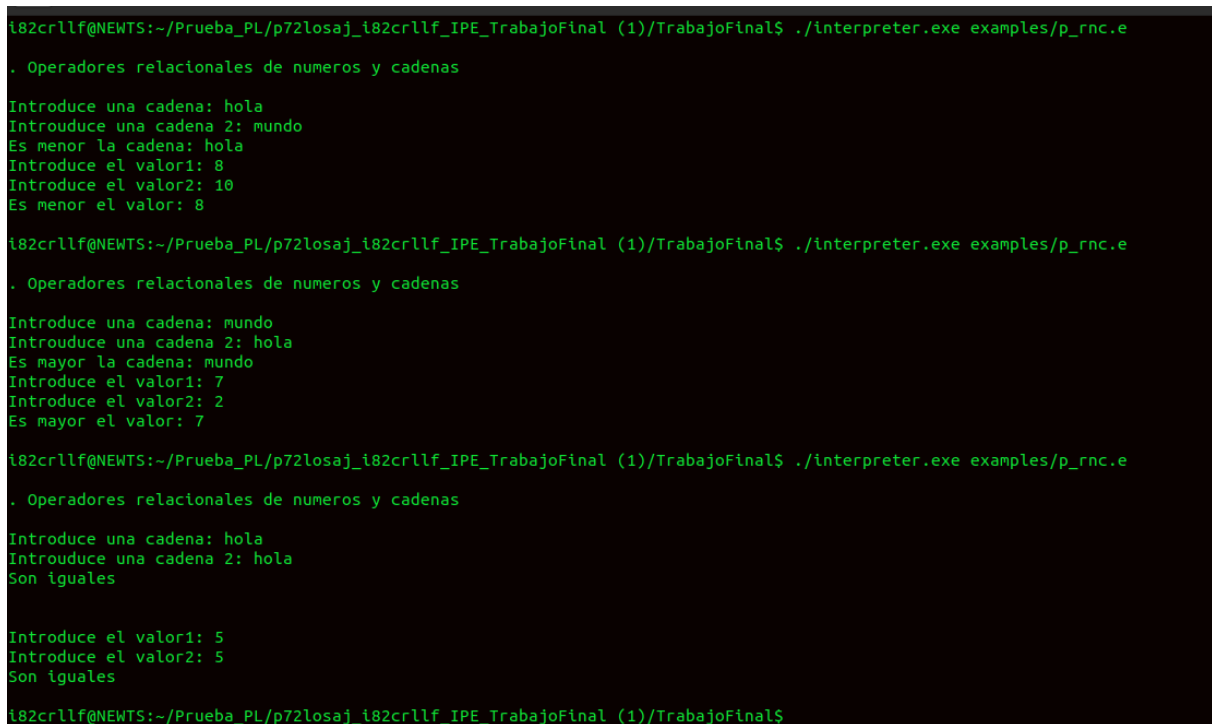
Elige una opcion:
6
Introduce el valor1: 99
Introduce el valor2: 4
Resultado de la operacion modulo es: 3
```

-test-error.txt: este fichero sirve para comprobar si lee bien la extensión del fichero y en caso de no ser la esperada muestra un mensaje de error.



```
Terminal
<abajoFinal (1)/TrabajoFinal$ ./interpreter.exe examples/test-error.txt
Programa:
Linea de error 1 --> Error,fichero con extension erronea:
examples/test-error.txt
i82crllf@NEWS:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$
```

-p\_rnc: este fichero sirve para comprobar los operadores relacionales.



```
i82crllf@NEWS:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$ ./interpreter.exe examples/p_rnc.e
. Operadores relacionales de numeros y cadenas
Introduce una cadena: hola
Introduce una cadena 2: mundo
Es menor la cadena: hola
Introduce el valor1: 8
Introduce el valor2: 10
Es menor el valor: 8

i82crllf@NEWS:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$ ./interpreter.exe examples/p_rnc.e
. Operadores relacionales de numeros y cadenas
Introduce una cadena: mundo
Introduce una cadena 2: hola
Es mayor la cadena: mundo
Introduce el valor1: 7
Introduce el valor2: 2
Es mayor el valor: 7

i82crllf@NEWS:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$ ./interpreter.exe examples/p_rnc.e
. Operadores relacionales de numeros y cadenas
Introduce una cadena: hola
Introduce una cadena 2: hola
Son iguales

Introduce el valor1: 5
Introduce el valor2: 5
Son iguales
i82crllf@NEWS:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$
```

-p\_bi: este fichero contiene una serie de bucles para la comprobación de bucles normales e inversos y un ejemplo de error en caso de bucle infinito.

```
i82crllf@NEWTs:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$ ./interpreter.exe examples/p_b1.e
prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

prueba bucle inverso con paso negativo

-----

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

prueba bucle inverso con paso positivo

-----
```

```
-----

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

prueba bucle normal

-----
```

```
Programa: ./interpreter.exe
Línea de error 23 --> Bucle infinito para:
    el valor desde es menor que el valor hasta y el incremento es negativo
i82crllf@NEWTs:~/Prueba_PL/p72losaj_i82crllf_IPE_TrabajoFinal (1)/TrabajoFinal$
```

# Capítulo 11

## CONCLUSIONES

### 11.1. PUNTOS FUERTES Y PUNTOS DÉBILES DEL INTÉRPRETE DESARROLLADO

En lo referente a los puntos fuertes, creemos que uno de nuestros puntos fuertes de nuestra practica es que es capaz de evitar cascadas de errores así como también que es capaz de avisar de la ubicación del error. Otro de los que hemos considerado puntos fuertes sería que puede analizar un número elevado de sentencias gracias a la implementación de sus reglas.

En lo referente a los puntos débiles, creemos que uno de ellos sería, por ejemplo, el hecho de que si espera un paréntesis izquierdo avisa del error pero no es capaz de solucionarlo por si mismo. Otro punto débil sería que no hemos conseguido implementar en las sentencias de control, la sentencia de selección múltiple.

### 11.2. REFLEXIÓN FINAL SOBRE EL TRABAJO REALIZADO

Según hemos observado, hemos sido capaces de llevar a cabo casi todas las tareas que se pedían en el control de tareas. Sin embargo, como se ha mencionado en el apartado anterior, no hemos podido llevar a cabo la tarea de realizar la sentencia de selección múltiple

de de las sentencias de control. No obstante hemos reunido en un fichero, "PruebasSM", lo que hemos hecho respecto a la selección múltiple, pero que por falta de tiempo y errores surgidos no hemos podido implementar en el practica.

## Capítulo 12

# BIBLIOGRAFÍA

Guía de Flex y Bison: [http://webdiis.unizar.es/asignaturas/LGA/material\\_2004\\_2005/Intro\\_Flex\\_Bison.pdf](http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf)

# Capítulo 13

## ANEXOS

Diagrama de la tabla de símbolos

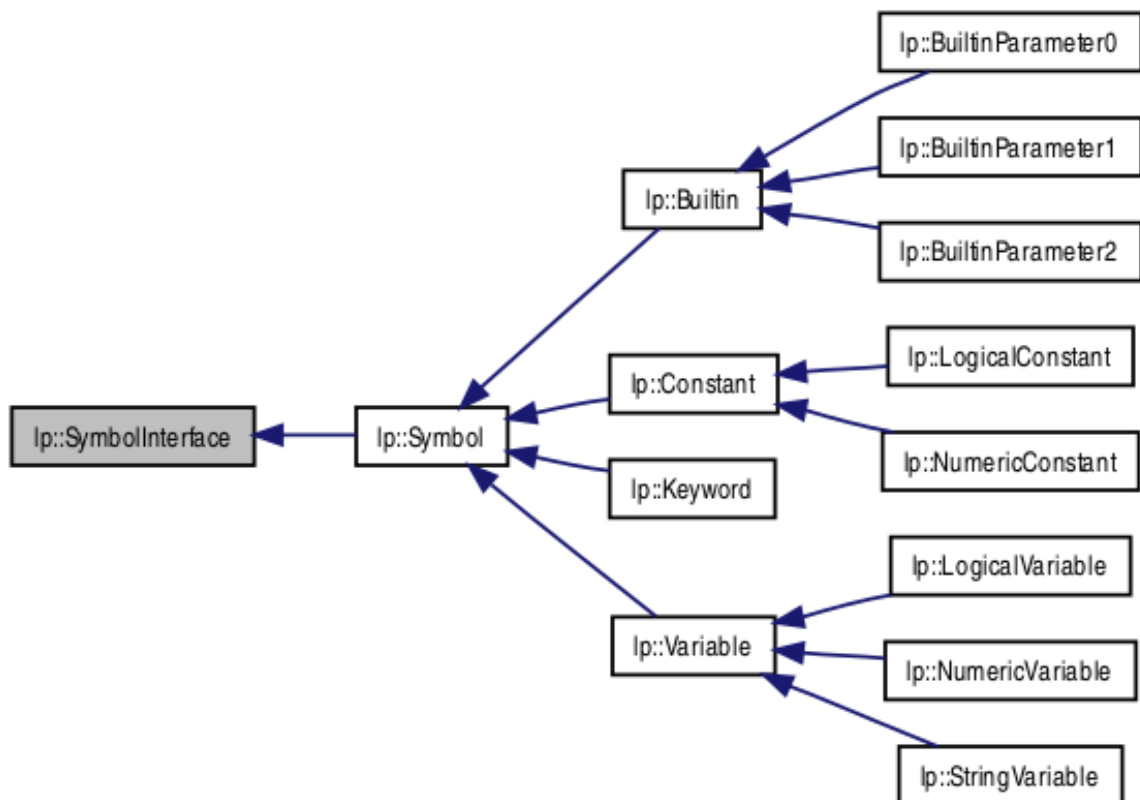


Diagrama de la clase ExpNode

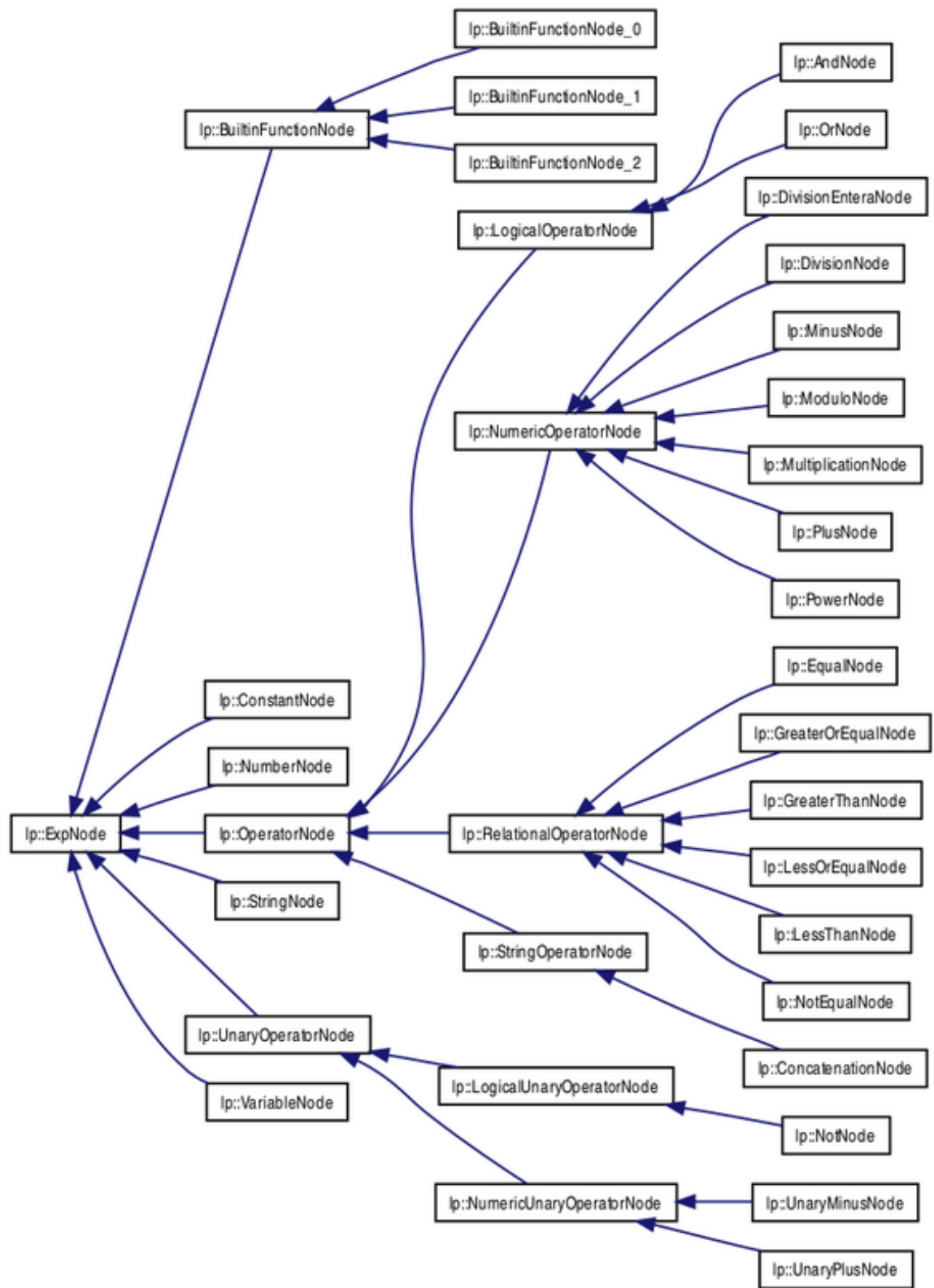


Diagrama de la clase Statement



