

# Procesadores de Lenguaje



## TRABAJO FINAL DE PRÁCTICAS

Damián Martínez Ávila

Universidad de Córdoba

Escuela Politecnica Superior de Córdoba

Grado de Ingenieria Informatica. Mencion en

Computación

3º Curso 2ºCuatrimestre

Córdoba a 26 de Junio de 2022

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Lenguaje de pseudocódigo</b>	<b>4</b>
2.1. Componentes Léxicos . . . . .	4
2.1.1. Palabras Reservadas . . . . .	4
2.1.2. Identificadores . . . . .	4
2.1.3. Números . . . . .	5
2.1.4. Cadenas . . . . .	5
2.1.5. Operador de asignación . . . . .	5
2.1.6. Operadores aritméticos . . . . .	6
2.1.7. Operador alfanumérico . . . . .	6
2.1.8. Operadores relacionales . . . . .	6
2.1.9. Operadores lógicos . . . . .	6
2.1.10. Comentarios . . . . .	7
2.1.11. Punto y Coma . . . . .	7
2.2. Sentencias . . . . .	7
2.2.1. Asignación . . . . .	7
2.2.2. Lectura . . . . .	7
2.2.3. Escritura . . . . .	7
2.2.4. Sentencias de control . . . . .	8
2.2.5. Comandos especiales . . . . .	8
<b>3. Tabla de Símbolos</b>	<b>9</b>
<b>4. Análisis léxico</b>	<b>9</b>
4.1. Definiciones de expresiones regulares . . . . .	9
4.2. Estados del analizador léxico . . . . .	9
4.3. Componentes Léxicos . . . . .	10
<b>5. Análisis sintáctico</b>	<b>11</b>
5.1. Simbolos de la gramática . . . . .	11
5.1.1. Símbolos terminales . . . . .	11
5.1.2. Símbolos no terminales . . . . .	12

5.2. Reglas de producción . . . . .	13
5.3. Acciones Semánticas . . . . .	15
<b>6. AST</b>	<b>30</b>
<b>7. Funciones auxiliares</b>	<b>34</b>
<b>8. Modo de obtención del intérprete</b>	<b>35</b>
<b>9. Modo de ejecución</b>	<b>38</b>
9.1. Modo interactivo . . . . .	38
9.2. Ejecución desde un fichero . . . . .	38
<b>10. Ejemplos</b>	<b>39</b>
10.1. binario.e . . . . .	39
10.2. conversion.e . . . . .	39
10.3. menu.e . . . . .	41
10.4. numeropalindromo.e . . . . .	41
10.5. wordle.e . . . . .	41
<b>11. Conclusiones</b>	<b>43</b>

## 1. Introducción

Para la realización del trabajo de final de prácticas de la asignatura de Procesadores de Lenguaje vamos a utilizar flex y bison para elaborar un intérprete de pseudocódigo en español

Contendra los siguientes apartados:

- 2) Características del lenguaje de pseudocódigo
- 3) Implementación de la tabla de símbolos
- 4) Descripción de los componentes léxicos y de sus expresiones regulares.
- 5) Descripción de la gramática de contexto libre
- 6) Implementación de AST
- 7) Resumen de las funciones auxiliares codificadas
- 8) Estructura del código correspondiente al intérprete
- 9) Distintos modos de ejecución del intérprete
- 10) Distintos ejemplos para testear nuestro intérprete
- 11) Conclusiones

## 2. Lenguaje de pseudocódigo

### 2.1. Componentes Léxicos

#### 2.1.1. Palabras Reservadas

Las palabras reservadas no se podrán utilizar como identificadores y no se distinguirá entre mayúsculas ni minúsculas.

Sentencias de control
leer
leer_cadena
escribir
escribir_cadena
si
entonces
si_no
fin_si
mientras
hacer
fin_mientras
repetir
hasta
para
fin_para
desde
paso
segun
valor
defecto
fin_segun

Constantes y operadores lógicos
verdadero
falso
o
y
no

Manejo de la pantalla
borrar_pantalla
lugar

#### 2.1.2. Identificadores

Los identificadores estarán compuestos por una serie de letras, dígitos y el subrayado. Deben comenzar por una letra y no podrán acabar con el símbolo

de subrayado, ni tener dos subrayados seguidos. En cuanto a las letras, no se distinguirá entre mayúsculas ni minúsculas.

Identificadores válidos	Identificadores no válidos
dato	_dato
dato_1	dato_
dato_1_a	dato__1

### 2.1.3. Números

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

### 2.1.4. Cadenas

Estará compuesta por una serie de caracteres delimitados por comillas simples:

‘Ejemplo de cadena’

‘Ejemplo de cadena con salto de línea

n y tabulador

t’

Deberá permitir la inclusión de la comilla simple utilizando la barra ( ): ‘Ejemplo de cadena con ’ comillas’ simples’.

Notas: Las comillas exteriores no se almacenarán como parte de la cadena. Por tanto, al escribir una cadena, no se deberán imprimir las comillas exteriores pero sí se deberán interpretar los comandos

n,

t y .

### 2.1.5. Operador de asignación

Descripción	Operador	Ejemplo
Asignación	<code>:=</code>	<code>dato := 1</code>

**2.1.6. Operadores aritméticos**

Descripcion	Operador	Ejemplo
Suma Unaria	+	+1
Suma Binaria	+	1+1
Resta Unaria	-	-1
Resta Binaria	-	1-1
Producto	*	1*1
Division	/	1/1
Division Entera	//	2//1
Modulo	%	8%2
Potencia	**	2**3

**2.1.7. Operador alfanumérico**

Descripcion	Operador	Ejemplo
Concatenación		'Hello'    'World'

**2.1.8. Operadores relacionales**

Descripcion	Operador	Ejemplo
Menor que	<	1 < 2
Menor o igual que	<=	1 <= 1
Mayor que	>	2 > 1
Mayor o igual que	>=	4 >= 3
Igual que	=	1=1
Distinto que	<>	1 <> 2

**2.1.9. Operadores lógicos**

Descripcion	Operador	Ejemplo
Disyunción lógica	o	1 < 3 o 4 > 2
Conjunción lógica	y	1 < 3 y 4 > 2
Negación lógica	no	no <i>dato</i> < 3

**2.1.10. Comentarios**

Descripcion	Operador
Comentario de varias lineas	<< Comentario de varias lineas >>
Comentario de una linea	# Comentario de una linea

**2.1.11. Punto y Coma**

Se utilizará para indicar el fin de una sentencia.

**2.2. Sentencias****2.2.1. Asignación**

Asignación
identificador := expresión numérica
identificador := expresión alfanumérica

**2.2.2. Lectura**

Descripcion	Sentencia
Declara a identificador como una variable numérica y le asigna el número leído.	Leer (identificador)
Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).	Leer_cadena (identificador)

**2.2.3. Escritura**

Descripcion	Sentencia
El valor de la expresión numérica es escrito en la pantalla.	Escribir (expresión numérica)
La cadena (sin comillas exteriores) es escrita en la pantalla.	Escribir_cadena (expresión alfanumérica)



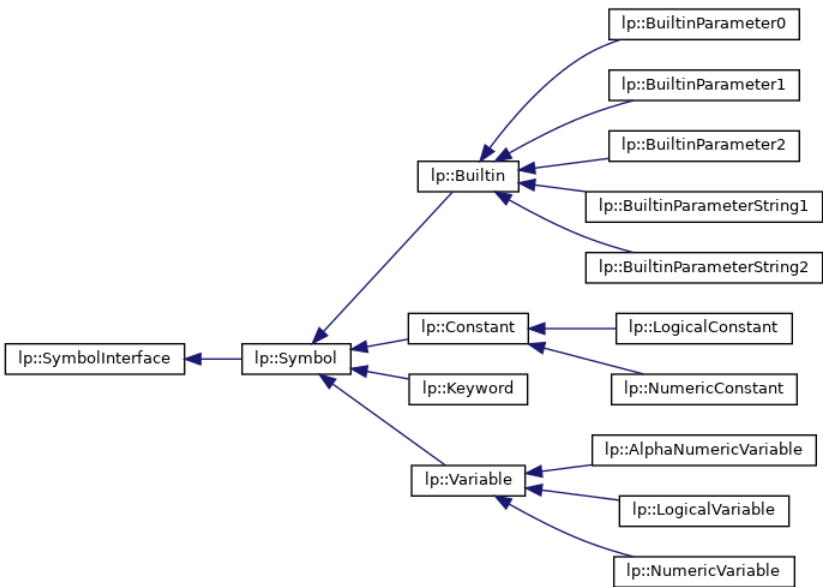
**2.2.4. Sentencias de control**

Descripcion	Sentencia
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

**2.2.5. Comandos especiales**

Descripcion	Sentencia
Borra la pantalla	borrar_pantalla
Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.	lugar(expresión, expresión)

3. Tabla de Símbolos



4. Análisis léxico

4.1. Definiciones de expresiones regulares

Nombre	Expresion Regular
DIGIT	[0-9]
LETTER	[a-zA-Z]
SUBRAYADO	[ _ ]
NUMBER1	{DIGIT}+\.?
NUMBER2	{DIGIT}*\.{DIGIT}+
NUMBER3	{DIGIT}+(\.{DIGIT}+)?([Ee][+ -]?{DIGIT}+)?
IDENTIFIER	{LETTER}({LETTER} {DIGIT} {SUBRAYADO}) ( {LETTER} {DIGIT} ))*

4.2. Estados del analizador léxico

- Estado\_Cadena
- Estado\_Comentario

- Estado\_Comentario2
- Error

### 4.3. Componentes Léxicos

Componente	Descripcion	Expresion Regular
;	Punto y Coma	"."
,	Coma	","
no	Operador lógico negación	"no"
o	Operador lógico disyunción	"o"
y	Operador lógico conjunción	"y"
Number	Número Entero/Flotante	{NUMBER1} {NUMBER2}
PowerNumber	Notación Científica	{NUMBER3}
Identifier	Identificador/Palabra Clave	{IDENTIFIER}
-	Resta	" - "
+	Suma	" + "
	Multiplicación	" * "
/	División	" / "
//	División Entera	" // "
(	Parentesis Izquierdo	" ("
)	Parentesis Derecho	" ) "
%	Modulo	" % "
*	Potencia	" * * "
:=	Asignación	" := "
=	Igualdad	" = "
<>	Distinto que	" < > "
>=	Mayor o igual que	" > = "
<=	Menor o igual que	" < = "
>	Mayor que	" > "
<	Menor que	" < "
{	Llave Izquierda	" { "
}	Llave Derecha	" } "
	Concatenación	"    "

## 5. Análisis sintáctico

### 5.1. Simbolos de la gramática

#### 5.1.1. Símbolos terminales

Mínima Precedencia	Máxima Precedencia	Asociatividad Izq	Sin Asociatividad
PRINT	ASSIGNMENT	OR	GREATER_OR_EQUAL
PRINT_STRING	POWER	AND	LESS_OR_EQUAL
READ		NOT	GREATER_THAN
READ_STRING		PLUS	LESS_THAN
IF		MINUS	EQUAL
THEN		CONCAT	NOT_EQUAL
ELSE		MULTIPLICATION	UNARY
WHILE		DIVISIONENTERA	
ENDIF		MODULO	
DO		DIVISION	
ENDWHILE		LPAREN	
REPEAT		RPAREN	
TILL			
CLEAR			
PLACE			
FOR			
STEP			
FROM			
ENDFOR			
LEFTCURLYBRACKET			
RIGHTCURLYBRACKET			
COMMA			
NUMBER			
STRING			
BOOL			

VARIABLE
UNDEFINED
CONSTANT
BUILTIN
SEMICOLON

### 5.1.2. Símbolos no terminales

program
stmtlist
stmt
controlSymbol
if
while
repeat
for
cond
assgn
print
print_string
read
read_string
keywords
clear
place
exp
listOfExp
restOfListOfExp

## 5.2. Reglas de producción

program: stmtlist
stmtlist: epsilon stmtlist: stmtlist stmt stmtlist: stmtlist error
stmt: SEMICOLON stmt: assign SEMICOLON stmt: print SEMICOLON stmt: print_string SEMICOLON stmt: read SEMICOLON stmt: read_string SEMICOLON stmt: clear SEMICOLON stmt: place SEMICOLON stmt: if stmt: while stmt: repeat stmt: for
controlSymbol: epsilon
if: IF controlSymbol cond THEN stmtlist ENDIF if: IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF
while: WHILE controlSymbol cond DO stmtlist ENDWHILE
repeat: REPEAT stmtlist TILL controlSymbol cond
for: FOR controlSymbol VARIABLE FROM exp TILL exp DO stmtlist ENDFOR for : FOR controlSymbol VARIABLE FROM exp TILL exp STEP exp DO stmtlist ENDFOR
cond: LPAREN exp RPAREN
asn: VARIABLE ASSIGNMENT exp asn: VARIABLE ASSIGNMENT asn asn: CONSTANT ASSIGNMENT exp asn: CONSTANT ASSIGNMENT asn asn: keywords ASSIGNMENT exp asn: keywords ASSIGNMENT asn

print: PRINT keywords SEMICOLON print: PRINT LPAREN keywords RPAREN SEMICOLON print: PRINT exp
print_string: PRINT_STRING keywords SEMICOLON print_string: PRINT_STRING LPAREN keywords RPAREN SEMICOLON print_string: PRINT_STRING exp
read: READ LPAREN VARIABLE RPAREN SEMICOLON read: READ LPAREN CONSTANT RPAREN SEMICOLON read: READ LPAREN keywords RPAREN SEMICOLON
read_string: READ_STRING LPAREN VARIABLE RPAREN SEMICOLON read_string: READ_STRING LPAREN CONSTANT RPAREN SEMICOLON read_string: READ_STRING LPAREN keywords RPAREN SEMICOLON
clear: CLEAR
place: PLACE LPAREN exp COMMA exp RPAREN
exp: NUMBER exp: STRING exp: exp PLUS exp exp: exp MINUS exp exp: exp MULTIPLICATION exp exp: exp DIVISION exp exp: exp DIVISIONENTERA exp exp: LPAREN exp RPAREN exp: PLUS exp %prec UNARY exp: MINUS exp %prec UNARY exp: exp MODULO exp exp: exp POWER exp exp: exp CONCAT exp exp: VARIABLE exp: CONSTANT exp: BUILTIN LPAREN listOfExp RPAREN exp: exp GREATER_THAN exp exp: exp GREATER_OR_EQUAL exp: exp LESS_THAN exp exp: exp LESS_OR_EQUAL exp exp: exp EQUAL exp exp: exp NOT_EQUAL exp exp: exp AND exp exp: exp OR exp exp: NOT exp

listOfExp: epsilon
listOfExp: exp restOfListOfExp
resOfListOfExp: epsilon
resOfListOfExp: COMMA exp restOfListOfExp

### 5.3. Acciones Semánticas

```

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

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

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

```

```

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

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

    // Control the interactive mode of execution of the interpreter
    if (interactiveMode == true && control == 0)
    {
        for(std::list<lp::Statement *>::iterator it = $$->begin();
            it != $$->end();
            it++)
        {
            (*it)->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: SEMICOLON /* Empty statement: ";" */
{
    // Create a new empty statement node
    $$ = new lp::EmptyStmt();
}
| asgn SEMICOLON
{
    // Default action
    // $$ = $1;
}
| print SEMICOLON
{
    // Default action
    // $$ = $1;
}
| print_string SEMICOLON
{
    // Default action
    // $$ = $1;
}
| read SEMICOLON
{
    // Default action
    // $$ = $1;
}
| read_string SEMICOLON
{
    // Default action
    // $$ = $1;
}
| clear SEMICOLON
{
    // Default action
    // $$ = $1;
}
```

```
| place SEMICOLON
| {
|     // Default action
|     // $$ = $1;
| }
/* NEW in example 17 */
| if
| {
|     // Default action
|     // $$ = $1;
| }
/* NEW in example 17 */
| while
| {
|     // Default action
|     // $$ = $1;
| }
| repeat
| {
|     // Default action
|     // $$ = $1;
| }
| for
| {
|     // Default action
|     // $$ = $1;
| }
```

```

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

/* NEW in example 17 */
if: /* Simple conditional statement */
IF controlSymbol cond THEN stmtlist ENDIF
{
    // Create a new if statement node
    $$ = new lp::IfStmt($3, $5);

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

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

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

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

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

repeat: REPEAT stmtlist TILL controlSymbol cond
{
    // Create a new while statement node
    $$ = new lp::RepeatTillStmt($5, $2);

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

```

```

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

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

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

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

/* NEW in example 17 */
cond: LPAREN exp RPAREN
    {
        $$ = $2;
    }
;

```

```

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

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

    | 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);
    }

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

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

```

```

print: PRINT keywords SEMICOLON
{
    execerror("Semantic error in \"read statement\": it is not allowed to print a keyword ",$2);
}

| PRINT LPAREN keywords RPAREN SEMICOLON
{
    execerror("Semantic error in \"read statement\": it is not allowed to print a keyword ",$3);
}

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

;

print_string: PRINT_STRING keywords SEMICOLON
{
    execerror("Semantic error in \"read statement\": it is not allowed to print a keyword ",$2);
}

| PRINT_STRING LPAREN keywords RPAREN SEMICOLON
{
    execerror("Semantic error in \"read statement\": it is not allowed to print a keyword ",$3);
}

| PRINT_STRING exp
{
    // Create a new print node
    $$ = new lp::PrintStringStmt($2);
}

;

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

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

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

;

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

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

| READ_STRING LPAREN keywords RPAREN SEMICOLON
{
    execerror("Semantic error in \"read statement\": it is not allowed to modify a keyword ",$3);
}

;

```

keywords:

```
PRINT {}  
| READ {}  
| IF {}  
| THEN {}  
| ELSE {}  
| WHILE {}  
| ENDIF {}  
| DO {}  
| ENDWHILE {}  
| REPEAT {}  
| TILL {}  
| CLEAR {}  
| PLACE {}  
| FOR {}  
| STEP {}  
| FROM {}  
| ENDFOR {}
```

;

clear: CLEAR

```
{  
    // Create a new print node  
    $$ = new lp::ClearStmt();  
}
```

;

place: PLACE LPAREN exp COMMA exp RPAREN

```
{  
    // Create a new print node  
    $$ = new lp::PlaceStmt($3,$5);  
}
```

;

```
exp:  NUMBER
    {
        // Create a new number node
        $$ = new lp::NumberNode($1);
    }

    |  STRING
    {
        // Create a new alphanumeric node
        $$ = new lp::AlphaNumericNode($1);
    }

    |  exp PLUS exp
    {
        // Create a new plus node
        $$ = new lp::PlusNode($1, $3);
    }

    |  exp MINUS exp
    {
        // Create a new minus node
        $$ = new lp::MinusNode($1, $3);
    }

    |  exp MULTIPLICATION exp
    {
        // Create a new multiplication node
        $$ = new lp::MultiplicationNode($1, $3);
    }

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

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

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

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

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

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

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



```
| exp CONCAT exp
| {
|   // Create a new power node
|   $$ = new lp::ConcatNode($1, $3);
| }
| VARIABLE
| {
|   // Create a new variable node
|   $$ = new lp::VariableNode($1);
| }
| CONSTANT
| {
|   // Create a new constant node
|   $$ = new lp::ConstantNode($1);
| }
```

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

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

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

                // Create a new Builtin Function with 1 parameter node

                if(e->getType()==NUMBER){
                    $$ = new lp::BuiltinFunctionNode_1($1,e);
                }

                if(e->getType()==UNDEFINED){
                    $$ = new lp::BuiltinFunctionStringNode_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

    if(e1->getType()==NUMBER){
        $$ = new lp::BuiltinFunctionNode_2($1,e1,e2);
    }

    if(e1->getType()==UNDEFINED){
        $$ = new lp::BuiltinFunctionStringNode_2($1,e1,e2);
    }
}
break;

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

```
| exp GREATER_THAN exp
| {
|   // Create a new "greater than" node
|   $$ = new lp::GreaterThanNode($1,$3);
| }
| exp GREATER_OR_EQUAL exp
| {
|   // Create a new "greater or equal" node
|   $$ = new lp::GreaterOrEqualNode($1,$3);
| }
| exp LESS_THAN exp
| {
|   // Create a new "less than" node
|   $$ = new lp::LessThanNode($1,$3);
| }
| exp LESS_OR_EQUAL exp
| {
|   // Create a new "less or equal" node
|   $$ = new lp::LessOrEqualNode($1,$3);
| }
| exp EQUAL exp
| {
|   // Create a new "equal" node
|   $$ = new lp::EqualNode($1,$3);
| }
```

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

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

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

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

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

| exp restOfListOfExp
{
    $$ = $2;

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

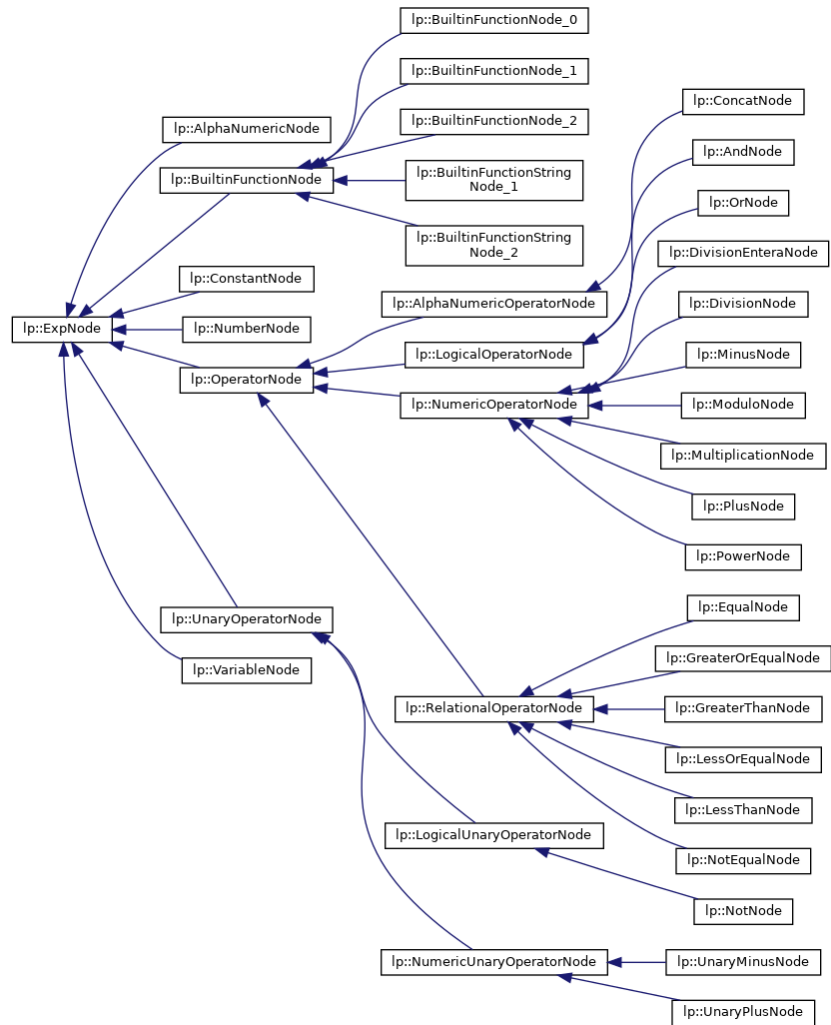
;
```

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

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

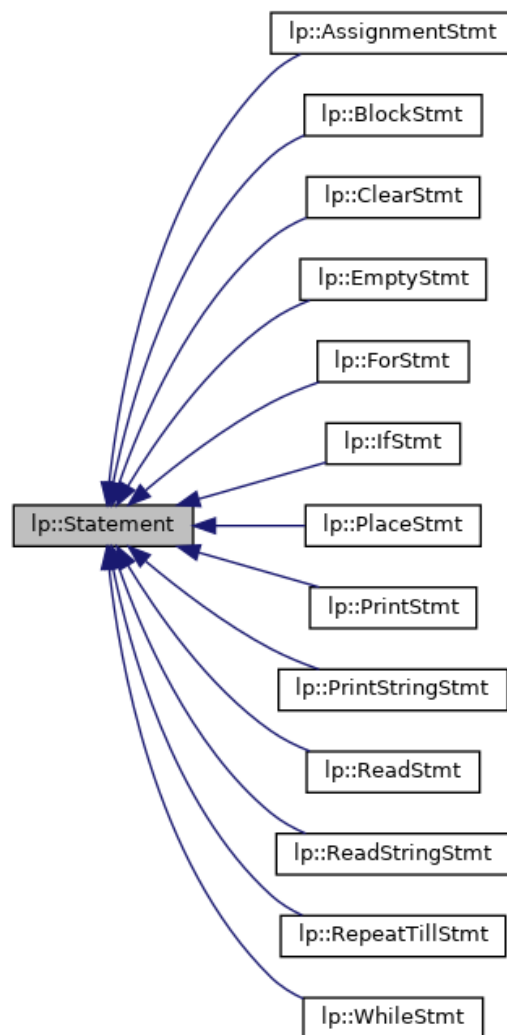
        // Insert the expression in the list of expressions
        $$->push_front($2);
    }
;
```

## 6. AST



Clase	Descripción
ExpNode	Nodo para una expresión
VariableNode	Nodo para una variable
ConstantNode	Nodo para una constante
NumberNode	Nodo para un número
AlphaNumericNode	Nodo para una cadena
UnaryOperatorNode	Nodo para un operador unario
NumericUnaryOperatorNode	Nodo para un operador unario numérico
LogicalUnaryOperatorNode	Nodo para un operador unario lógico
UnaryMinusNode	Nodo para una resta unaria
UnaryPlusNode	Nodo para una suma unaria
OperatorNode	Nodo para un operador
NumericOperatorNode	Nodo para un operador numérico
AlphaNumericOperatorNode	Nodo para un operador alfanumérico
ConcatNode	Nodo para una concatenación
RelationalOperatorNode	Nodo para un operador relacional
LogicalOperatorNode	Nodo para un operador lógico
PlusNode	Nodo para una suma
MinusNode	Nodo para una resta
MultiplicationNode	Nodo para una multiplicación
DivisionNode	Nodo para una división
DivisionEnteraNode	Nodo para una división entera
ModuloNode	Nodo para un módulo
PowerNode	Nodo para una potencia
BuiltinFunctionNode	Nodo para una función integrada
BuiltinFunctionNode_0	Nodo para una función integrada sin parámetros
BuiltinFunctionNode_1	Nodo para una función integrada con un parámetro
BuiltinFunctionNode_2	Nodo para una función integrada con dos parámetros
BuiltinFunctionNodeString_1	Nodo para una función integrada con un parámetro
BuiltinFunctionNodeString_2	Nodo para una función integrada con dos parámetros
GreaterThanNode	Nodo para el operador relacional mayor que
GreaterOrEqualNode	Nodo para el operador relacional mayor o igual que
LessThanNode	Nodo para el operador relacional menor que
LessOrEqualNode	Nodo para el operador relacional menor o igual que
EqualNode	Nodo para el operador relacional igual que
NotEqualNode	Nodo para el operador relacional distinto de
AndNode	Nodo para el operador lógico y
OrNode	Nodo para el operador lógico o
NotNode	Nodo para el operador lógico no





Clase	Descripción
Statement	Clase raíz para sentencias
AssignmentStmt	Sentencia de asignación
ClearStmt	Sentencia para limpiar la pantalla
EmptyStmt	Sentencia vacía
ForStmt	Sentencia para el bucle Para
IfStmt	Sentencia Condicional
PlaceStmt	Sentencia para lugar_pantalla
PrintStmt	Sentencia para escribir
PrintStringStmt	Sentencia para escribir_cadena
ReadStmt	Sentencia para leer
ReadStringStmt	Sentencia para leer_cadena
RepeatTillStmt	Sentencia para bucle hasta
WhileStmt	Sentencia para bucle mientras

## 7. Funciones auxiliares

Aparte de las funciones matemáticas de las que ya disponíamos se han implementado las siguientes funciones:

Como funciones BuiltIn para variables alfanuméricas:

- `getlength(alphanumericVariable)`: Nos devuelve la longitud de una cadena.

```
int GetLength(char * value)
{
    return strlen(value);
}
```

- `getnletter(alphanumericVariable, Number)`: Nos devuelve el carácter que se encuentre en la posición que nos dice el segundo parámetro.

```
char* GetNLetter(char * value, int pos)
{
    char * result= new char[2];
    result[0] = value[pos-1];
    result[1] = '\0';
    return result;
}
```

Para comprobar si un número está representado con notación científica se usa la siguiente función auxiliar:

```
bool is_number(const std::string& s)
{
    std::string::const_iterator it = s.begin();
    while (it != s.end() && std::isdigit(*it)) ++it;
    return !s.empty() && it == s.end();
}
```

Dicha función la usamos para la sentencia `leer` para comprobar que realmente es un valor válido para un número con notación científica.

## 8. Modo de obtención del intérprete

El proyecto consta de los siguientes directorios:

- **ast**: Contiene los ficheros de implementación para el ast.
  - **ast.cpp**: Código fuente del ast.
  - **ast.hpp**: Cabecera del ast.
  - **makefile**: Configuración de make. Sirve para la compilación.
- **error**: Contiene los ficheros de la implementación de las funciones que ayudan al control de errores.
  - **error.cpp**: Código fuente de errores.
  - **error.hpp**: Cabecera de errores.
  - **makefile**: Configuración de make. Sirve para la compilación.
- **examples**: Contiene ficheros con pseudocódigo de ejemplo para probar el intérprete. Los ejemplos serán explicados de forma individual más adelante en el documento.
  - **binario.e**
  - **conversion.e**
  - **menu.e**
  - **numeropalindromo.e**
  - **wordle.e**
- **html**: Documentación generada por doxygen.
- **includes**: Contiene un fichero para macros. del proyecto.
  - **macros.hpp**: Contiene macros que facilitan la lectura del código.
- **parser**: Contiene los ficheros de los analizadores léxico y sintáctico.
  - **interpreter.l**: Código fuente del analizador léxico
  - **interpreter.y**: Código fuente del analizador sintáctico.
  - **makefile**: Configuración de make. Sirve para la compilación.

- table: Contiene los ficheros de implementación de la tabla de símbolos.
  - alphaNumericFunction.cpp: Código fuente de funciones alfanuméricas.
  - alphaNumericFunction.hpp: Cabecera de funciones alfanuméricas.
  - alphaNumericVariable.cpp: Fichero fuente de la clase alphaNumericVariable.
  - alphaNumericVariable.hpp: Cabecera de la clase alphaNumericVariable.
  - builtin.cpp: Código fuente de la clase Builtin.
  - builtin.hpp: Cabecera de la clase Builtin.
  - builtinParameter0.cpp: Código fuente de la clase BuiltinParameter0.
  - builtinParameter0.hpp: Cabecera de la clase BuiltinParameter0.
  - builtinParameter1.cpp: Código fuente de la clase BuiltinParameter1.
  - builtinParameter1.hpp: Cabecera de la clase BuiltinParameter1.
  - builtinParameter2.cpp: Código fuente de la clase BuiltinParameter2.
  - builtinParameter2.hpp: Cabecera de la clase BuiltinParameter2.
  - builtinParameterString1.cpp: Código fuente de la clase BuiltinParameterString1.
  - builtinParameterString1.hpp: Cabecera de la clase BuiltinParameterString1.
  - builtinParameterString2.cpp: Código fuente de la clase BuiltinParameterString2.
  - builtinParameterString2.hpp: Cabecera de la clase BuiltinParameterString2.
  - constant.cpp: Código de fuente de la clase Constant.
  - constant.hpp: Cabecera de la clase Constant.
  - init.cpp: Código fuente para inicializar la tabla de símbolos.
  - init.hpp: Cabecera de la inicialización de la tabla de símbolos.
  - keyword.cpp: Código fuente de la clase Keyword.

- keyword.hpp: Cabecera de la clase Keyword.
- logicalConstant.cpp: Código fuente de la clase LogicalConstant.
- logicalConstant.hpp: Cabecera de la clase LogicalConstant.
- logicalVariable.cpp: Código fuente de la clase LogicalVariable.
- logicalVariable.hpp: Cabecera de la clase LogicalVariable.
- mathFunction.cpp: Código fuente de funciones matemáticas.
- mathFunction.hpp: Cabecera de funciones matemáticas.
- numericConstant.cpp: Código fuente de la clase NumericConstant.
- numericConstant.hpp: Cabecera de la clase NumericConstant.
- numericVariable.cpp: Código fuente de la clase NumericVariable.
- numericVariable.hpp: Cabecera de la clase NumericVariable.
- symbol.cpp: Código fuente de la clase Symbol.
- symbol.hpp: Cabecera de la clase Symbol.
- symbolInterface.hpp: Cabecera de la clase SymbolInterface.
- table.cpp: Código fuente de la clase Table.
- table.hpp: Cabecera de la clase Table.
- tableInterface.hpp: Cabecera de la clase TableInterface.
- variable.cpp: Código fuente de la clase Variable.
- variable.hpp: Cabecera de la clase Variable.

Los archivos que se encuentran en el directorio raíz son:

- Doxyfile: Archivo para generar la documentación de doxygen.
- interpreter.cpp: Código fuente del intérprete.
- makefile: Archivo para compilar el programa.

## 9. Modo de ejecución

### 9.1. Modo interactivo

Se ejecutarán las instrucciones tecleadas desde un terminal de texto. Se utilizará el carácter de fin de fichero para terminar la ejecución: Control + D

Ejemplo de ejecución: `interpreter.exe`

### 9.2. Ejecución desde un fichero

Se interpretarán las sentencias de un fichero pasado como argumento desde la línea de comandos. El fichero deberá tener la extensión “.e”.

Ejemplo de ejecución: `interpreter.exe example.e`

## 10. Ejemplos

### 10.1. binario.e

Programa que convierte un número entero a código binario.

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

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

r := ' ';

mientras (n <> 0) hacer
    si ((n % 2) = 0) entonces
        #concatenación con 0
        r := '0' || r;

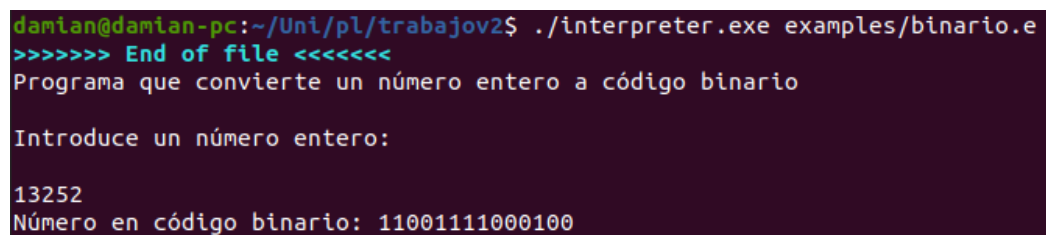
    si_no
        #concatenación con 1
        r := '1' || r;

    fin_si

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

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

Ejemplo de ejecución:



```
damian@damian-pc:~/Uni/pl/trabajov2$ ./interpreter.exe examples/binario.e
>>>>>> End of file <<<<<<
Programa que convierte un número entero a código binario

Introduce un número entero:

13252
Número en código binario: 11001111000100
```

### 10.2. conversion.e

Programa de ejemplo de conversión de tipo. Se comprueba la conversión de tipo de una variable.



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

borrar_pantalla;

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

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

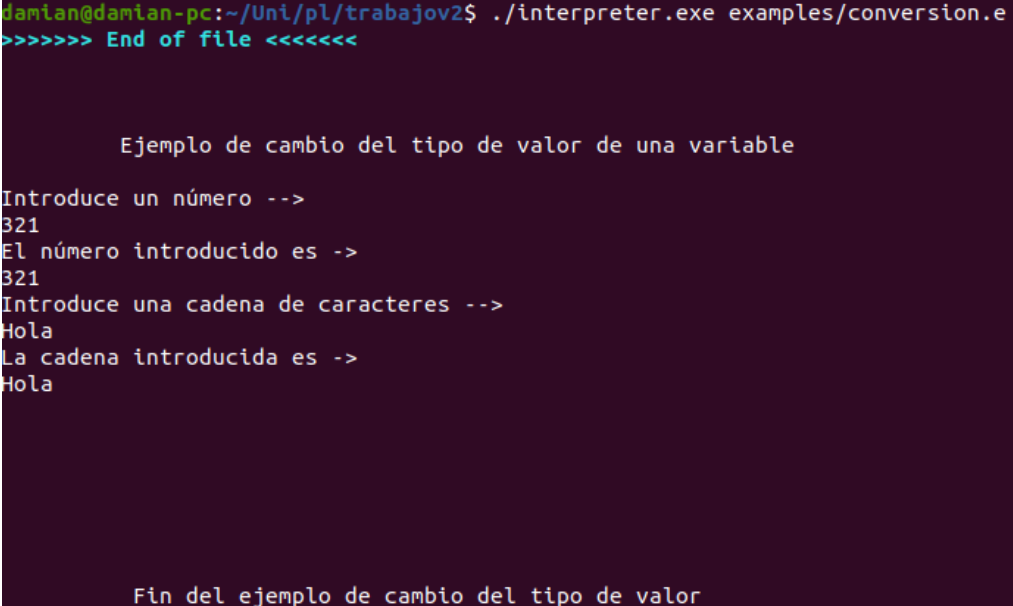
escribir_cadena('El número introducido es -> ');
escribir(dato);

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

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

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

Ejemplo de ejecución:



```
damian@damian-pc:~/Uni/pl/trabajov2$ ./interpreter.exe examples/conversion.e
>>>>>> End of file <<<<<<

      Ejemplo de cambio del tipo de valor de una variable

Introduce un número -->
321
El número introducido es ->
321
Introduce una cadena de caracteres -->
Hola
La cadena introducida es ->
Hola

      Fin del ejemplo de cambio del tipo de valor
```

### 10.3. menu.e

Programa main con dos opciones: factorial y máximo común divisor

### 10.4. numeropalindromo.e

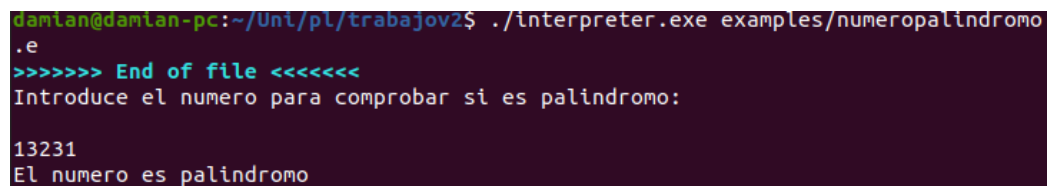
Programa que comprueba si un numero es palindromo

```
#Programa que identifica si un numero es palindromo

escribir_cadena('Introduce el numero para comprobar si es palindromo: \n');
leer(numero);
temp:=numero;
sum:=0;
mientras (numero>0) hacer
    r:=numero%10;
    sum:=(sum*10)+r;
    numero:=numero//10;
fin_mientras
si (temp=sum) entonces
    escribir_cadena('El numero es palindromo');
si_no
    escribir_cadena('El numero no es palindromo');

fin_si
```

Ejemplo de ejecución:



```
damian@damian-pc:~/Uni/pl/trabajov2$ ./interpreter.exe examples/numeropalindromo
.e
>>>>>> End of file <<<<<<
Introduce el numero para comprobar si es palindromo:
13231
El numero es palindromo
```

### 10.5. wordle.e

Programa que simula el juego Wordle (<https://wordle.danielfrg.com/>)

Ejemplo de ejecución:

```
damian@damian-pc:~/Uni/pl/trabajov2$ ./interpreter.exe examples/wordle.e
>>>>>> End of file <<<<<<
Wordle

Introduce la palabra a encontrar (max 10 caracteres)):

palabra
El numero de intentos es:
5
Introduzca su intento:
potable

Palabra Errónea

1
Carácter Correcto

2
Carácter Incorrecto

3
Carácter Incorrecto

4
Carácter Correcto

5
Carácter Correcto

6
Carácter Correcto Pero En Posición Errónea

7
Carácter Incorrecto

Numero de Intentos Restantes:
4

Introduzca su intento:
palabra
Palabra Acertada
```

## 11. Conclusiones

Como conclusiones de este trabajo me gustaria destacar como me ha hecho ver el funcionamiento de un compilador sin que haya sido excesivamente complicado.

Es cierto que podría haber mejorado ciertas cosas como por ejemplo las BuiltInFunctions que no estoy seguro de que las haya implementado correctamente con parametros alfanuméricos, llegar a implementar la sentencia segun o añadir que se pudieran concatenar variables de varios tipos que hubiese venido bien para que el ultimo ejemplo se viera mejor.