Analizador sintáctico

Procesadores de lenguajes

Ingeniería Informática - Especialidad de Computación

Tercer curso, segundo cuatrimestre

Escuela Politécnica Superior de Córdoba

Universidad de Córdoba

Curso académico: 2022 - 2023

Córdoba a 28 de junio de 2023

Manuel Casas Castro -31875931R - i72cascm@uco.es

Sergio Lucena Téllez - 49505734M - i72lutes@uco.es

Índice

1. INTRODUCCIÓN

1.1 Breve descripción del trabajo

En este proyecto, desarrollaremos un intérprete de pseudocódigo en español utilizando los lenguajes Flex y Bison. Aplicaremos los conocimientos adquiridos en el curso de procesadores de lenguajes para llevar a cabo el análisis léxico y sintáctico. El objetivo es crear un intérprete que pueda ejecutar programas escritos en pseudocódigo, brindando una herramienta en español que facilite la programación en este lenguaje específico.

1.2 Partes del documento

- **Lenguaje de pseudocódigo:** un lenguaje diseñado específicamente para este proyecto, en el cual se encuentran los elementos léxicos y las sentencias utilizadas.

- **Tabla de símbolos:** una estructura que muestra y almacena información relevante del programa, como variables, funciones, constantes, entre otros.

- **Análisis léxico**: una fase crucial donde se crean los tokens que el intérprete reconocerá, identificando y clasificando los componentes léxicos presentes en el código fuente.

- **Análisis sintáctico:** se reciben los tokens generados en el análisis léxico y se verifica si cumplen las reglas sintácticas establecidas por nuestro lenguaje, asegurando la coherencia y la estructura adecuada del programa.

- **AST (Árbol Sintáctico Abstracto):** una representación estructurada del programa en forma de árbol, que se genera durante el análisis sintáctico y almacena la estructura lógica de las clases utilizadas en nuestro intérprete.

- **Modos de ejecución:** se explican las distintas formas en las que el intérprete puede ser ejecutado, brindando opciones flexibles para adaptarse a las necesidades y preferencias de los usuarios.

- **Ejemplos:** una recopilación de ejemplos creados tanto por el profesor como por nosotros mismos, diseñados para realizar pruebas exhaustivas en nuestro intérprete y validar su correcto funcionamiento en diferentes escenarios.

- **Conclusiones:** una reflexión final sobre el trabajo realizado, destacando los aspectos positivos y negativos que surgieron durante el desarrollo del proyecto, y ofreciendo una visión general sobre los logros alcanzados y las posibles áreas de mejora.

2. LENGUAJE DE PSEUDOCÓDIGO

2.1 Componentes léxicos

2.1.1 Palabras reservadas

Las palabras mencionadas están previamente establecidas en nuestra tabla de símbolos, mientras que los demás componentes léxicos están definidos en el intérprete. Además de esto, en la tabla de símbolos también se incluirán las constantes numéricas y las funciones.

Estructura para las keywords:

```
static struct {
        std::string name ;
        int token;
        } keyword[] = {
                        {"and",  AND},
                        {"or",  OR},
                        {"not",  NOT},
                        {"read",  READ},
                        {"read_string",  READSTRING},
                        {"print", PRINT},
                        {"print_string",  PRINTSTRING},
                        {"if",    IF},
                        {"then",  THEN},
                        {"else",  ELSE},
                        {"end_if",  ENDIF},
                        {"while", WHILE},
                        {"do",  DO},
                        {"end_while",  ENDWHILE},
                        {"repeat",  REPEAT},
                        {"until",  UNTIL},
                        {"for",  FOR},
                        {"end_for",  ENDFOR},
                        {"from",  FROM},
                        {"step",  STEP},
                        {"to",  TO},
                        {"case",  CASE},
                        {"value",  VALUE},
                        {"break", BREAK},
                        {"default",  DEFAULT},
                        {"end_case",  ENDCASE},
                        {"clear_screen",  CLEARSCREEN},
                        {"place",  PLACE},
                        {"",      0}
                       };
```

2.1.2 Identificadores

La identificación de este componente léxico se basará en el uso de expresiones regulares. Está conformado por una secuencia de letras, dígitos y subrayados.

Debe iniciar con una letra y no puede terminar con un subrayado, ni tener dos subrayados consecutivos.

Expresión regular y su controlador:

```
DIGIT  [0-9]

LETTER [a-zA-Z]

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

2.1.3 Número

La identificación de este componente léxico se realiza mediante el uso de expresiones regulares y permite reconocer números enteros, números reales de punto fijo y números reales con notación científica. Todos estos números son tratados de manera uniforme

Expresión regular y su controlador:

```
DIGIT  [0-9]

NUMBER {DIGIT}{DIGIT}*(\.{DIGIT}{DIGIT}*(([Ee][+\-]?)?{DIGIT}{DIGIT}*)?)?
```

2.1.4 Cadena

La identificación de este componente léxico se realiza mediante el uso de expresiones regulares y consiste en reconocer una serie de caracteres delimitados por comillas simples. Es posible incluir comillas simples dentro del texto mediante ciertas reglas establecidas.

Expresión regular y su controlador:

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

2.1.5 Operador asignación

Mediante el operador de asignación :=, tenemos la capacidad de asignar un valor a un identificador.

Línea de código de la asignación:

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

2.1.6 Operadores aritméticos

Los operadores numéricos sólo se aplicarán a variables numéricas, no se pueden utilizar en variables no numéricas ni operar con dos variables de tipos diferentes.

```
"-"   { return MINUS;  }

"+"   { return PLUS;   }

"*"     { return MULTIPLICATION; }

"/"     { return DIVISION; }

"//"  { return DIV_INT; }

"%"     { return MODULO; }

"^"     { return POWER; }
```

2.1.7 Operadores alfanuméricos

El operador de concatenación **“ || ”** tiene la misma funcionalidad que el operador de suma, pero se utiliza específicamente para variables de tipo cadena. Este componente léxico será representado dentro del intérprete.

Línea de código de la concatenación:

```
"||"  { return CONCAT; }
```

2.1.8 Operadores relaciones de números y cadenas

Los operadores relacionales funcionarán tanto para variables numéricas como para variables alfanuméricas de forma independiente. Ambos retornarán y tendrán la misma funcionalidad que en C++.

```
"="   { return EQUAL; }

"<>"  { return NOT_EQUAL; }

">="  { return GREATER_OR_EQUAL; }

"<="  { return LESS_OR_EQUAL; }

">"   { return GREATER_THAN; }

"<"   { return LESS_THAN; }
```

2.1.9 Operadores lógicos

Los operadores lógicos se emplean para conectar diferentes valores, identificadores o fórmulas, y su resultado será verdadero o falso según se cumplan o no las condiciones establecidas.

Definición de los operadores lógicos:

```
static struct {
        std::string name ;
        int token;
        } keyword[] = {
                        {"and",  AND},
                        {"or",  OR},
                        {"not",  NOT},
```

2.1.10 Comentarios

Los comentarios de una sola línea y de múltiples líneas se gestionan a través de estados de autómatas desde el analizador léxico de la siguiente manera:

Estructura de los comentarios de línea múltiples:

```
"<<"                                    {
                                            yymore();
                                            BEGIN(COMENTARIO_LINEAS_MULTIPLES);
                                        }


<COMENTARIO_LINEAS_MULTIPLES>\n    {
                                    lineNumber++;
                                    yymore();
                                }

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


<COMENTARIO_LINEAS_MULTIPLES>">>"   {
                                        BEGIN(INITIAL);
                                    }
```

Estructura de los comentarios de líneas simples:

```
#                                   {
                                        yymore();
                                        BEGIN(COMENTARIO_LINEA_SIMPLE);
                                    }

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

<COMENTARIO_LINEA_SIMPLE>\n         {
                        lineNumber++;
                                        BEGIN(INITIAL);
                                    }
```

2.1.11 Punto y coma

El token punto y coma es utilizado para separar las diferentes líneas de código reconocidas en el analizador sintáctico:

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

2.2 Sentencias

2.2.1 Asignación

Para llevar a cabo una asignación, utilizaremos el mismo operador tanto para expresiones numéricas como para expresiones alfanuméricas.

- Para expresiones numéricas, la sintaxis será la siguiente:

  identificador := expresión numérica

Se declara el identificador como una variable numérica y se le asigna el valor de la expresión numérica. Las expresiones numéricas se formarán utilizando números, variables numéricas y operadores numéricos.

- Para expresiones alfanuméricas, la sintaxis será la siguiente:

  identificador := expresión alfanumérica

Se declara el identificador como una variable alfanumérica y se le asigna el valor de la expresión alfanumérica.Las expresiones alfanuméricas se formarán utilizando números, variables alfanuméricas y el operador de concatenación (||).

2.2.2 Lectura

Para llevar a cabo la lectura, utilizaremos dos funciones distintas, ya que no podemos asignar un tipo de dato al identificador de antemano. A continuación, se detallan las funciones de lectura:

Declara el identificador como una variable numérica y le asigna el número leído.
read(identificador)

Declara el identificador como una variable alfanumérica y le asigna la cadena leída (sin comillas).
read_string(identificador)

2.2.3 Escritura

Para la escritura, solo necesitaremos utilizar una única función. Inicialmente, existían dos funciones, pero resultaban redundantes, ya que si analizamos el tipo de dato dentro de la expresión de asignación, podemos escribirlo de una forma u otra.

El valor será mostrado en pantalla.
print(número);

La cadena será mostrada en pantalla.
print_string(cadena);

En caso de ser una expresión alfanumérica, se permitirá la interpretación de comandos de salto de línea y tabuladores que puedan aparecer.

2.2.4 Sentencias de control

En esta sección, se presentarán los pseudocódigos de las sentencias que posteriormente serán analizadas en la sección de análisis léxico.

- Sentencia condicional simple

```
if condición
        then lista de sentencias
end_if
```

- Sentencia condicional compuesta

```
if condición
        then lista de sentencias
        else lista de sentencias
end_if
```

- Bucle “while”

```
while condición do
        lista de sentencias
end_while
```

- Bucle “repeat”

```
repeat
        lista de sentencias
until condición
```

- Bucle “for”

```
for identificador
        from expresión numérica 1
        to expresión numérica 2
        [step expresión numérica 3]
        do
                lista de sentencias
end_for
```

- Sentencia “case”

```
case (expresión)
        value expresión 1: lista de sentencias
        value expresión 2: lista de sentencias
        …
 [default: lista de sentencias]
end_case
```

2.2.5 Comandos especiales

- clear_screen

  borra la pantalla

- place(expresión numérica1, expresión numérica2)

Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

3. TABLA DE SÍMBOLOS

3.1 Descripción

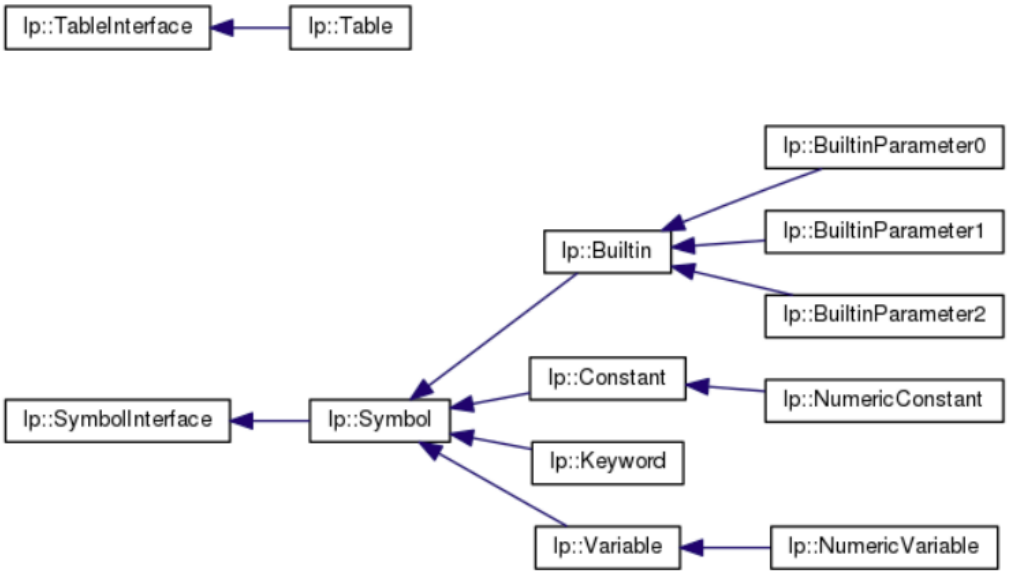
La tabla de símbolos es una estructura de datos en forma de mapa donde cada uno de los datos generados en nuestro programa se clasificará y almacenará en ella. Dentro de nuestra tabla, podemos encontrar diferentes tipos de datos que se vuelven cada vez más específicos.

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

Clase "SymbolInterface":

- Clase "Symbol":
    - Clase "Builtin": Define una función predefinida.
        - Clase "BuiltinParameter0": Define una función predefinida sin parámetros.
        - Clase "BuiltinParameter1": Define una función predefinida con un parámetro.
        - Clase "BuiltinParameter2": Define una función predefinida con dos parámetros.
    - Clase "Constant": Define las constantes predefinidas.
        - Clase "LogicalConstant": Clase para las constantes lógicas.
        - Clase "NumericConstant": Clase para las constantes numéricas.
    - Clase "Keyword": Clase que almacena las palabras reservadas predefinidas.
    - Clase "Variable": Clase que define las variables generadas en la tabla de símbolos.
        - Clase "LogicalVariable": Clase que define las variables como lógicas.
        - Clase "NumericVariable": Clase que define las variables como numéricas.
        - Clase "AlphanumericVariable": Clase que define las variables como cadenas.
- Clase "TableInterface": Clase abstracta con las funciones para acceder a la tabla de símbolos.
    - Clase "Table": Clase que define la tabla de símbolos.

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



Estructura de la clase "SymbolInterface"

4. ANÁLISIS LÉXICO

4.1 Descripción

El inicio del proceso de nuestro intérprete consiste en leer el código fuente de manera secuencial, caracter por caracter, con el objetivo de identificar los componentes léxicos y enviarlos posteriormente al analizador sintáctico. En esta sección se describen detalladamente todos los componentes léxicos y las expresiones regulares que los representan.

Estructura del analizador léxico:

```
[ \t]   { ; }    /* skip white space and tabular */

\n      {
            /* Line counter */
            lineNumber++;
            /* return NEWLINE; */
        }

";"     {
            return SEMICOLON;
        }


","     {
            return COMMA;
        }

"-"     { return MINUS;  }

"+"     { return PLUS;   }

"*"     { return MULTIPLICATION; }
```

```
"/"     { return DIVISION; }

"("     { return LPAREN; }

")"     { return RPAREN; }

"//"    { return DIV_INT; }

"%"     { return MODULO; }

"^"     { return POWER; }

":="    { return ASSIGNMENT; }

"="     { return EQUAL; }

"<>"    { return NOT_EQUAL; }

">="    { return GREATER_OR_EQUAL; }

"<="    { return LESS_OR_EQUAL; }

">"     { return GREATER_THAN; }

"<"     { return LESS_THAN; }

":"     { return COLON; }

"["     { return SQUARE_LEFT_BRACKET; }

"]"     { return SQUARE_RIGHT_BRACKET; }

"||"    { return CONCAT; }

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

                 return NUMBER;
                }

(_{IDENTIFIER}|{IDENTIFIER}_) {
                                warning("Lexical error: bad identifier.","identifier");
                              }
```

```
{IDENTIFIER}        {

                    std::string identifier(yytext);
                     yylval.string =  strdup(yytext);

                    if (table.lookupSymbol(identifier) == false)
                       {
                           lp::NumericVariable *n = new lp::NumericVariable(identifier,VARIABLE,UNDEFINED,0.0);
                           table.installSymbol(n);
                           return VARIABLE;
                       }
                    else
                       {
                           lp::Symbol  *s = table.getSymbol(identifier);
                           return s->getToken();
                       }
                }


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

                yylval.string =  strdup(stringWithoutQuotes.c_str());
                return ALPHA;
}
```

5. Análisis sintáctico

5.1 Símbolos terminales

Los símbolos terminales representan los elementos específicos y concretos de un lenguaje de programación. Estos símbolos terminales son los que aparecen directamente en el código fuente y tienen un significado preciso. Pueden ser palabras clave, operadores, constantes o cualquier otro elemento del lenguaje que tenga una interpretación clara y definida. Los símbolos terminales son utilizados en las reglas de la gramática para reconocer y manipular las estructuras y expresiones válidas del lenguaje, permitiendo así su correcta interpretación y ejecución.

```
/* Minimum precedence */

/*****************************************/
%token SEMICOLON COLON SQUARE_LEFT_BRACKET SQUARE_RIGHT_BRACKET
/*****************************************/
%token CLEARSCREEN
/*****************************************/
%token PRINT PRINTSTRING READ READSTRING IF ELSE THEN ENDIF WHILE DO ENDWHILE REPEAT UNTIL FOR FROM TO STEP ENDFOR CASE VALUE BREAK DEFAULT ENDCASE PLACE
/*****************************************/
%right ASSIGNMENT
/*****************************************/
%token COMMA
/*****************************************/
%token <number> NUMBER
/*****************************************/
/*****************************************/
%token <logic> BOOL
/*****************************************/
%token <string> VARIABLE UNDEFINED CONSTANT BUILTIN ALPHA
/*****************************************/

/* No associativity */
%nonassoc GREATER_OR_EQUAL LESS_OR_EQUAL GREATER_THAN LESS_THAN  EQUAL NOT_EQUAL
/*****************************************/
%nonassoc  UNARY


/* Left associativity */

/*****************************************************/
%left CONCAT
/*****************************************/
%left OR
/*****************************************/
%left AND
/*****************************************/
%left NOT
/*****************************************************/
%left PLUS MINUS
/*****************************************/
%left MULTIPLICATION DIVISION DIV_INT MODULO
/*****************************************/
%left LPAREN RPAREN
/*****************************************/

// Maximum precedence
%right POWER
```

5.2 Símbolos no terminales

Los símbolos no terminales representan las partes más grandes y generales de una gramática. Son como etiquetas o nombres que se utilizan para definir reglas y estructuras gramaticales en un lenguaje de programación. Estos símbolos no terminales no representan

elementos concretos o finales, sino que son utilizados para construir y organizar las diferentes partes de un programa de manera coherente y comprensible.

```
/* Type of the non-terminal symbols */
%type <expNode> exp cond
/*****************************************/
%type <parameters> listOfExp  restOfListOfExp
/*****************************************/
%type <stmts> stmtlist
/*****************************************/
%type <st> stmt asgn print printString read readString if while repeat for clear case place
/*****************************************/
%type <cases> firstCase otherCases
/*****************************************/
%type <prog> program
/*****************************************/
```

5.3 Reglas de producción

Las reglas de producción son un conjunto de instrucciones que definen la estructura válida en una gramática específica. En las reglas de producción se describe cómo se pueden combinar los tokens provenientes del analizador léxico para formar construcciones gramaticales más complejas.

A continuación se muestran las reglas de producción utilizadas en nuestro analizador sintáctico:

SEMICOLON:

```
stmt: SEMICOLON  /* Empty statement: ";" */
      {
        // Create a new empty statement node
        $$ = new lp::EmptyStmt();
      }
    | asgn  SEMICOLON
      {}
    | print SEMICOLON
      {}
    | printString SEMICOLON
      {}
    | read SEMICOLON
      {}
    | readString SEMICOLON
      {}
```

```
    | if SEMICOLON
      {}
    | while SEMICOLON
      {}
    | repeat SEMICOLON
      {}
    | clear SEMICOLON
      {}
    | for SEMICOLON
      {}
    | place SEMICOLON
      {}
    | case SEMICOLON
    {}
;
```

WHILE:

```
while:  WHILE controlSymbol cond DO stmtlist ENDWHILE
```

IF:

```
if: IF controlSymbol cond THEN stmtlist ENDIF
| IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF
```

FOR:

```
for: FOR controlSymbol VARIABLE FROM exp TO exp DO stmtlist ENDFOR

FOR controlSymbol VARIABLE FROM exp TO exp STEP exp DO stmtlist ENDFOR
```

CASE:

```
case: CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET DEFAULT COLON stmtlist SQUARE_RIGHT_BRACKET ENDCASE

firstCase: VALUE NUMBER COLON stmtlist BREAK otherCases

otherCases: VALUE NUMBER COLON stmtlist BREAK otherCases
```

REPEAT:

```
repeat: REPEAT controlSymbol stmtlist UNTIL cond
```

CONDITION:

```
cond:   LPAREN exp RPAREN
```

ASSIGNMENT:

```
asgn:   VARIABLE ASSIGNMENT exp
      | VARIABLE ASSIGNMENT asgn
```

PRINT:

```
print:  PRINT LPAREN exp RPAREN
```

PRINT_STRING:

```
printString:  PRINTSTRING LPAREN exp RPAREN
```

READ:

```
read:  READ LPAREN VARIABLE RPAREN
```

READ_STRING:

```
readString:  READSTRING LPAREN VARIABLE RPAREN
```

CLEAR:

```
clear: CLEARSCREEN
```

PLACE:

```
place: PLACE LPAREN exp COMMA exp RPAREN
```

EXPRESSION:

```
exp:    NUMBER
    |   ALPHA
    |   exp PLUS exp
    |   exp MINUS exp
    |   exp MULTIPLICATION exp
    |   exp DIVISION exp
    |   LPAREN exp RPAREN
    |   PLUS exp %prec UNARY
    |   exp MODULO exp
    |   exp DIV_INT exp
    |   exp POWER exp
    |   VARIABLE
    |   CONSTANT
| BUILTIN LPAREN listOfExp RPAREN
    | exp GREATER_THAN exp
    | exp GREATER_OR_EQUAL exp
    | exp LESS_THAN exp
```

```
| exp LESS_OR_EQUAL exp

| exp EQUAL exp

| exp NOT_EQUAL exp

| exp AND exp

| exp OR exp

| NOT exp
```

LIST_OF_EXPRESSION:

```
listOfExp: //epsilon

| exp restOfListOfExp
```

REST_OF_LST_OF_EXPRESSION:

```
restOfListOfExp: //epsilon

| COMMA exp restOfListOfExp
```

5.4 Acciones semánticas:

A continuación se muestran las acciones semánticas de las palabras claves más importantes y cómo gestionan errores sintácticos o semánticos a través de las reglas de producción:

IF:

```
if: IF controlSymbol cond THEN stmtlist ENDIF{
      // Create a new if statement node
      $$ = new lp::IfStmt($3, $5);
      // To control the interactive mode
      control--;
    }
    | IF controlSymbol cond stmtlist ENDIF{
        execerror("Syntax error: missing THEN symbol.", "IF controlSymbol cond THEN stmtlist ENDIF");
    }
    | IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF{
      $$ = new lp::IfStmt($3, $5, $7);
      control--;
      }
    | IF controlSymbol cond stmtlist ELSE stmtlist ENDIF{
        execerror("Syntax error: missing THEN symbol.", "IF controlSymbol cond THEN stmtlist ENDIF");
    }
```

FOR:

```
for: FOR controlSymbol VARIABLE FROM exp TO exp DO stmtlist ENDFOR{
      $$ = new lp::ForStmt($3, $5, $7, $9);
      control--;
    }
    | FOR controlSymbol VARIABLE FROM exp TO exp STEP exp DO stmtlist ENDFOR{
      $$ = new lp::ForStmt($3, $5, $7, $9, $11);
      control--;
    }
    | FOR controlSymbol CONSTANT FROM exp TO exp STEP exp DO stmtlist ENDFOR{
      execerror("Semantic error in for statement: it is not allowed to modify a constant ", "FOR cont
    }
    | FOR controlSymbol CONSTANT FROM exp TO exp DO stmtlist ENDFOR{
      execerror("Semantic error in for statement: it is not allowed to modify a constant ", "FOR cont
    }
    | FOR controlSymbol VARIABLE exp TO exp STEP exp DO stmtlist ENDFOR{
      execerror("Syntax error: missing FROM symbol of for statement", "FOR controlSymbol VARIABLE FRC
    }
    | FOR controlSymbol VARIABLE FROM exp STEP exp DO stmtlist ENDFOR{
      execerror("Syntax error: missing TO symbol of for statement", "FOR controlSymbol VARIABLE FROM
    }
```

CASE:

```
case: CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET DEFAULT COLON stmtlist SQUARE_RIGHT_BRACKET ENDCASE{
        $$ = new lp::CaseStmt($4, $6, $10);
        control --;
      }
      | CASE controlSymbol exp RPAREN firstCase SQUARE_LEFT_BRACKET DEFAULT COLON stmtlist SQUARE_RIGHT_BRACKET ENDCASE{
          execerror("Syntax error: missing LPAREN symbol.", "CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET
      }
      | CASE controlSymbol LPAREN exp firstCase SQUARE_LEFT_BRACKET DEFAULT COLON stmtlist SQUARE_RIGHT_BRACKET ENDCASE{
          execerror("Syntax error: missing RPAREN symbol.", "CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET
      }
      | CASE controlSymbol LPAREN exp RPAREN firstCase DEFAULT COLON stmtlist SQUARE_RIGHT_BRACKET ENDCASE{
          execerror("Syntax error: missing SQUARE_LEFT_BRACKET symbol.", "CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_
      }
      | CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET COLON stmtlist SQUARE_RIGHT_BRACKET ENDCASE{
          execerror("Syntax error: missing DEFAULT symbol.", "CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET
      }
      | CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET DEFAULT stmtlist SQUARE_RIGHT_BRACKET ENDCASE{
          execerror("Syntax error: missing COLON symbol.", "CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET D
      }
      | CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE_LEFT_BRACKET DEFAULT COLON stmtlist ENDCASE{
          execerror("Syntax error: missing SQUARE_RIGHT_BRACKET symbol.", "CASE controlSymbol LPAREN exp RPAREN firstCase SQUARE
      }
;

firstCase: VALUE NUMBER COLON stmtlist BREAK otherCases{
             $$ = $6;
             lp::ExpNode *e = new lp::NumberNode($2);
             lp::Value *v = new lp::Value(e, $4);
             $$->push_front(v);
           }
           | VALUE COLON stmtlist BREAK otherCases{
               execerror("Syntax error: missing NUMBER.", "VALUE NUMBER COLON stmtlist BREAK otherCases");
           }
           | VALUE NUMBER stmtlist BREAK otherCases{
               execerror("Syntax error: missing COLON symbol.", "VALUE NUMBER COLON stmtlist BREAK otherCases");
           }
           | VALUE NUMBER COLON stmtlist otherCases{
               execerror("Syntax error: missing BREAK symbol.", "VALUE NUMBER COLON stmtlist BREAK otherCases");
           }
;

otherCases: VALUE NUMBER COLON stmtlist BREAK otherCases{
              $$ = $6;
              lp::ExpNode *e = new lp::NumberNode($2);
              lp::Value *v = new lp::Value(e, $4);
              $$->push_front(v);
            }
            |   //epsilon
            {
              $$ = new std::list<lp::Value *>();
            }
            | VALUE COLON stmtlist BREAK otherCases{
               execerror("Syntax error: missing NUMBER.", "VALUE NUMBER COLON stmtlist BREAK otherCases");
            }
            | VALUE NUMBER stmtlist BREAK otherCases{
               execerror("Syntax error: missing COLON symbol.", "VALUE NUMBER COLON stmtlist BREAK otherCases");
            }
            | VALUE NUMBER COLON stmtlist otherCases{
               execerror("Syntax error: missing BREAK symbol.", "VALUE NUMBER COLON stmtlist BREAK otherCases");
            }

repeat: REPEAT controlSymbol stmtlist UNTIL cond{
            $$ = new lp::RepeatStmt($5, $3);
            control--;
          }
          | REPEAT controlSymbol stmtlist cond{
              execerror("Syntax error: missing UNTIL symbol of do while statement", "REPEAT stmtlist UNTIL cond");
          }
;
```

6. AST(Árbol Sintáctico Abstracto)

6.1 Descripción

Dentro de la estructura del árbol, se encuentran las siguientes clases

- Clase "Statement":
    - "AskForKeyStmt": Permite interrumpir la ejecución del programa en modo interactivo.
    - "AssignmentStmt": Evalúa la función de asignación.
    - "RepeatStmt": Evalúa la función del bucle "repeat".
    - "EmptyStmt": Evalúa una función vacía.
    - "ForStmt": Evalúa la función del bucle "for".
    - "CaseStmt": Evalúa la función de la sentencia "case".
    - "IfStmt": Evalúa la función "if".
    - "PrintStmt": Evalúa la función "print".
    - "ReadStmt": Evalúa la función "read".
    - "ReadStringStmt": Evalúa la función "read_string".
    - "WhileStmt": Evalúa la función del bucle "while".

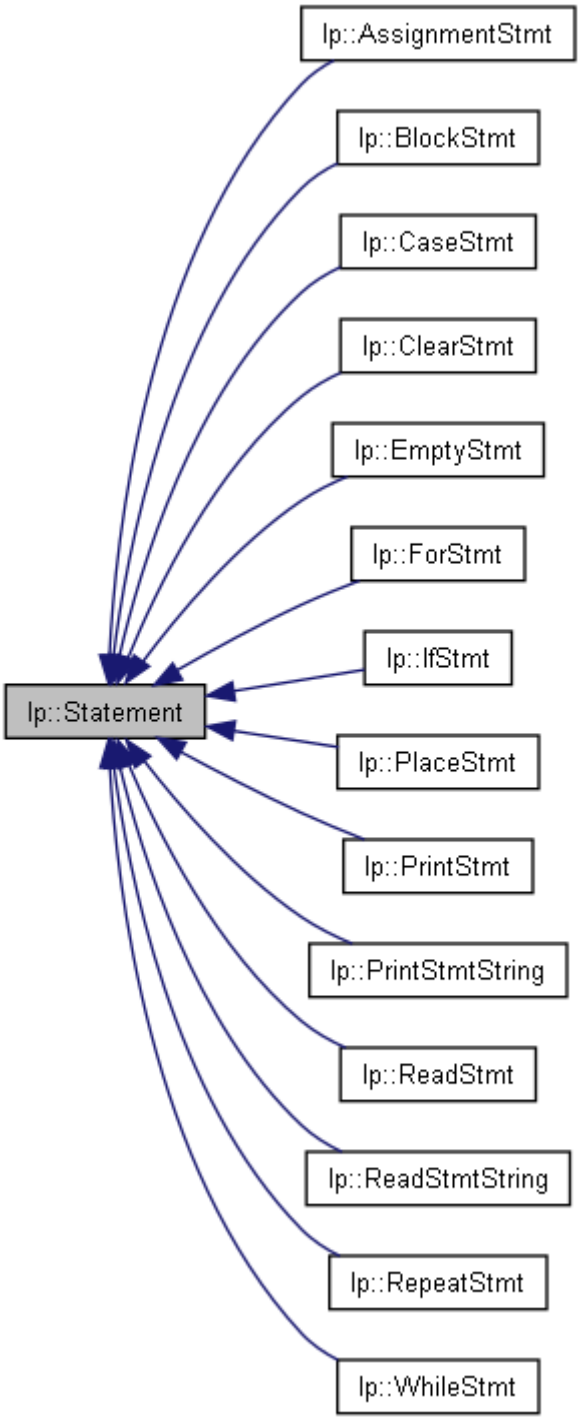
- Clase “ExpNode”:
    - Clase "BuiltinFunctionNode": Define los constructores de las funciones predefinidas.
        - Clase "BuiltinFunctionNode 0": Define los constructores de las funciones sin parámetros.
        - Clase "BuiltinFunctionNode 1": Define los constructores de las funciones con un único parámetro.
        - Clase "BuiltinFunctionNode 2": Define los constructores de las funciones con dos parámetros.
    - Clase "ConstantNode": Define las constantes en la tabla de símbolos.
    - Clase "NumberNode": Define las variables numéricas dentro de la tabla de símbolos.
    - Clase "OperatorNode": Define los operadores de las variables.
        - Clase "LogicalOperatorNode": Define los operadores lógicos que afectan a las variables.

        - Clase "AndNode": Define el operador lógico AND.
        - Clase "OrNode": Define el operador lógico OR.
    - Clase "NumericOperatorNode": Define los operadores para las variables numéricas.
        - Clase "DivisionNode": Define la operación de división entre dos variables numéricas.
        - Clase "IntegerDivisionNode": Define la operación de división entera entre dos variables numéricas.
        - Clase "MinusNode": Define la operación de resta entre dos variables numéricas.
        - Clase "ModuloNode": Define la operación de módulo entre dos variables numéricas.
        - Clase "MultiplicationNode": Define la operación de multiplicación entre dos variables numéricas.
        - Clase "PlusNode": Define la operación de suma entre dos variables numéricas.
        - Clase "PowerNode": Define la operación de potencia entre dos variables numéricas.
    - Clase "RelationalOperatorNode": Define los operadores para las variables numéricas o alfanuméricas que resultan en verdadero o falso.
        - Clase "EqualNode": Define la operación de igualdad para las variables numéricas o alfanuméricas.
        - Clase "GreaterOrEqualNode": Define la operación de mayor o igual que para las variables numéricas o alfanuméricas.
        - Clase "GreaterThanNode": Define la operación de mayor que para las variables numéricas o alfanuméricas.
        - Clase "LessOrEqualNode": Define la operación de menor o igual que para las variables numéricas o alfanuméricas.
        - Clase "LessThanNode": Define la operación de menor que para las variables numéricas o alfanuméricas.
        - Clase "NotEqualNode": Define la operación de desigualdad para las variables numéricas o alfanuméricas.

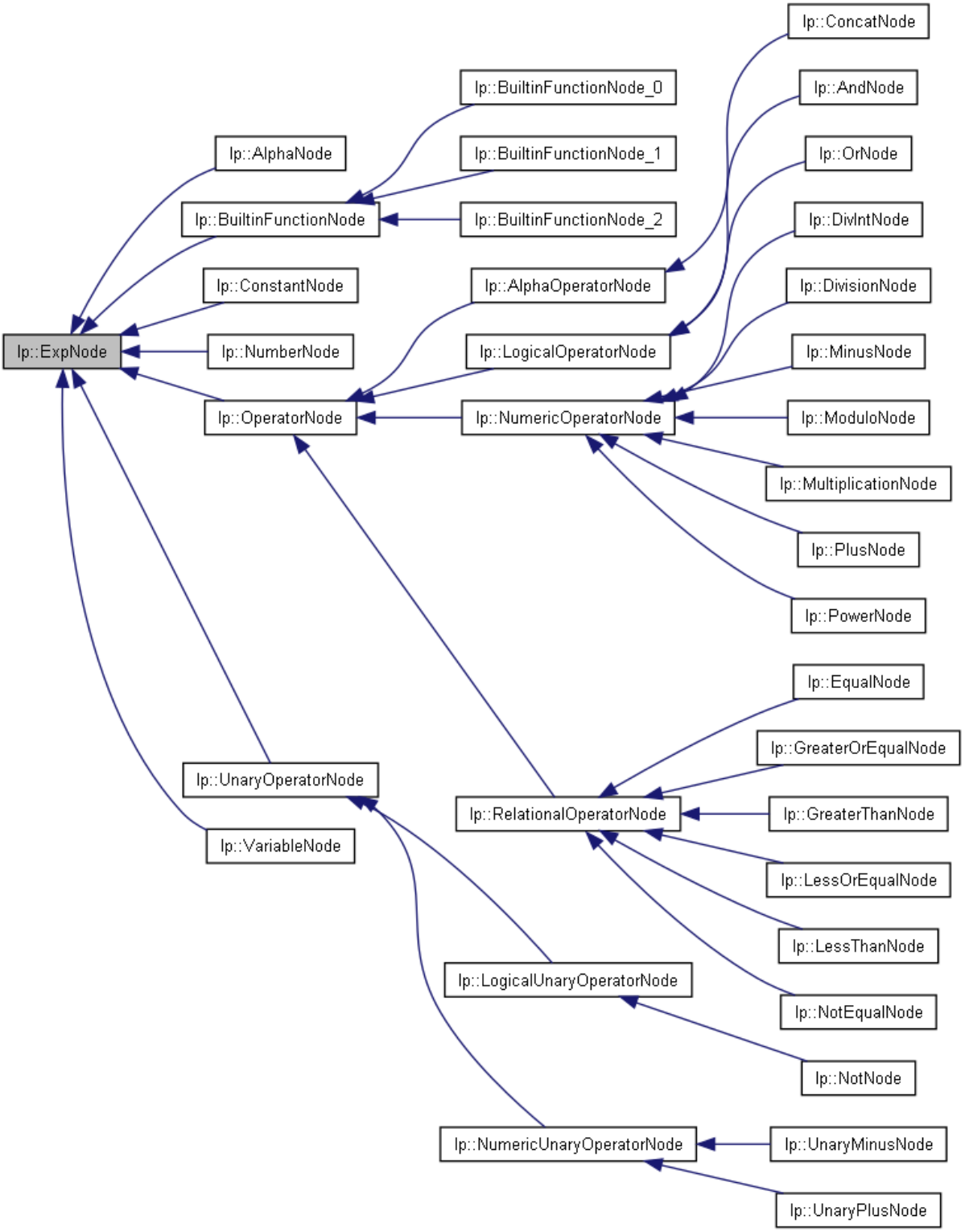
    - Clase "AlphaOperatorNode": Define los operadores específicos para las variables alfanuméricas.
        - Clase "ConcatenationNode": Define la función de concatenación para dos variables alfanuméricas.
- Clase "AlphaNode": Define las variables de cadena dentro de la tabla de símbolos.
- Clase "UnaryOperatorNode": Define los operadores unarios que afectan a las variables.
    - Clase "LogicalUnaryOperatorNode": Define los operadores unarios lógicos.
        - Clase "NotNode": Define el operador NOT.
    - Clase "NumericUnaryOperatorNode": Define los operadores unarios que afectan a las variables numéricas.
        - Clase "UnaryMinusNode": Define el operador unario de resta (-).
        - Clase "UnaryPlusNode": Define el operador unario de suma (+).

Clase "VariableNode": Define las variables que se generarán en la tabla de símbolos.

Estructuras de manera gráfica de la clase “Statement”:



Estructura gráficamente de la clase ExpNode:



7. MODO DE OBTENCIÓN DEL INTÉRPRETE

7.1 Descripción de los directorios y ficheros

El proyecto está estructurado en varios directorios y archivos:

- Subdirectorio "ast": Contiene los archivos relacionados con el árbol de sintaxis abstracta y las clases asociadas.
    - “ast.cpp": Implementación de las clases del árbol de sintaxis abstracta.
    - "ast.hpp": Declaración de las clases del árbol de sintaxis abstracta.
    - "makefile": Archivo para compilar el subdirectorio "ast".

- Subdirectorio "error": Incluye los archivos relacionados con el tratamiento de errores.
    - "error.cpp": Implementación de las funciones de recuperación de errores.
    - "error.hpp": Declaración de las funciones de recuperación de errores.
    - "makefile": Archivo para compilar el subdirectorio "error".

- Subdirectorio "examples": Contiene archivos de prueba.
    - "basic-tests.p": archivo de prueba para bucles básicos.
    - “binario.p”: archivo de prueba para conversión de número a binario
    - “conversion,p”: archivo de prueba para conversión de cadena a número de una variable.
    - “menu.p”: archivo de prueba para comprobar diferentes bucles
    - “menu2.p: archivo de prueba para comprobar diferentes bucles
    - “menu2_error.p”: archivo de prueba para comprobar diferentes errores
- Subdirectorio "includes": Contiene archivos para mejorar el programa.
    - "macros.hpp": Macros para mejorar la visualización en pantalla.

- Subdirectorio "parser": Incluye archivos para el análisis léxico y sintáctico.
    - "interpreter.l": Archivo Lex con las expresiones regulares del analizador léxico.
    - "interpreter.y": Archivo Yacc con la gramática del analizador sintáctico.
    - "makefile": Archivo para compilar el subdirectorio "parser".

- Subdirectorio "table": Contiene archivos relacionados con la tabla de símbolos.
    - "AlphanumericVariable.cpp": Implementación de las funciones de la clase "alphanumericVariable".
    - "AlphanumericVariable.hpp": Declaración de la clase "alphanumericVariable".
    - "builtin.cpp": Implementación de las funciones de creación de funciones predefinidas.
    - "builtin.hpp": Declaración de las clases de funciones predefinidas.
    - "builtinParameter0.cpp": Implementación de la función predefinida "builtinParameter0".
    - "builtinParameter0.hpp": Declaración de la clase "builtinParameter0".
    - "builtinParameter1.cpp": Implementación de la función predefinida "builtinParameter1".
    - "builtinParameter1.hpp": Declaración de la clase "builtinParameter1".
    - "builtinParameter2.cpp": Implementación de la función predefinida "builtinParameter2".
    - "builtinParameter2.hpp": Declaración de la clase "builtinParameter2".
    - "constant.cpp": Implementación de las funciones de la clase "constant" (heredada de la clase “symbol").
    - "constant.hpp": Declaración de la clase "constant".
    - "init.cpp": Implementación de las funciones de inicialización en la tabla de símbolos con las constantes predefinidas.
    - "init.hpp": Declaración de la clase "init".
    - "keyword.cpp": Implementación de la función de la clase "keyword".
    - “keyword.hpp": Declaración de la clase "keyword" (heredada de la clase "symbol").
    - “logicalConstant.cpp": Implementación de las funciones de la clase "logicalConstant".
    - “logicalConstant.hpp": Declaración de la clase "logicalConstant".
    - "logicalVariable.cpp": Implementación de las funciones de la clase "logicalVariable".
    - "logicalVariable.hpp": Declaración de la clase "logicalVariable".
    - "mathFunction.cpp": Implementación de las funciones matemáticas predefinidas.
    - "mathFunction.hpp": Declaración de las funciones matemáticas predefinidas.
    - "numericConstant.cpp": Implementación de las funciones de la clase "numericConstant".
    - "numericConstant.hpp": Declaración de la clase "numericConstant".

- "numericVariable.cpp": Implementación de las funciones de la clase "numericVariable".
- "numericVariable.hpp": Declaración de la clase "numericVariable".
- "symbol.cpp": Implementación de las funciones de la clase "symbol".
- "symbol.hpp": Declaración de la clase "symbol".
- "table.cpp": Implementación de las funciones de la clase "table".
- "table.hpp": Declaración de la clase "table".
- "tableInterface.hpp": Definición de la clase abstracta "tableInterface".
- "variable.cpp": Implementación de la función de la clase "variable".
- "variable.hpp": Declaración de las clases "variable" (heredada de la clase "symbol").

- "Doxyfile": Archivo que genera la documentación de los subdirectorios del proyecto.

- "interpreter.cpp": Programa principal de nuestro proyecto.

- "makefile": Archivo para compilar el intérprete.

8. MODO DE EJECUCIÓN

Nuestra aplicación ofrecerá múltiples modos de ejecución. Podremos interactuar con el programa en tiempo real a través del modo interactivo, lo que nos permitirá realizar acciones mientras el programa se ejecuta. También tendremos la opción de ejecutar el programa proporcionando un archivo como argumento con la extensión ".p", lo que facilitará la ejecución automática del programa con los comandos y datos previamente definidos en dicho archivo.

8.1 Interactiva

El modo de ejecución interactivo brinda al usuario la capacidad de utilizar el programa directamente desde la terminal. Gracias a este modo, podemos escribir nuestro código mientras el programa se ejecuta y también tenemos la posibilidad de interrumpir la ejecución en cualquier momento. Esto nos permite una interacción fluida y en tiempo real con el programa, facilitando la prueba y depuración de nuestro código durante su ejecución.

8.2 A partir de un fichero

En este modo, el usuario tiene la opción de pasar un archivo como argumento, el cual contendrá el pseudocódigo que deseamos ejecutar. Esta funcionalidad resulta muy útil al utilizar el intérprete, ya que nos permite trabajar con documentos más complejos de manera directa.

Para ejecutar nuestro programa, debemos seguir el siguiente formato:

./interprete.exe fichero.p

9. EJEMPLOS

9.1 basic-tests.p

```
read(variable);

if(variable = 0) then
    print_string('La variable es igual a cero.');
else
    print_string('La variable no es igual a cero.');
end_if;


N := 5;
i := 0;

for i from  1 to N step 1 do
    print_string('El bucle for se esta ejecutando');
end_for;

variable := 0;
while (variable <> 100) do
    variable := variable + 10;
end_while;

repeat
    read_string(cadena);
    print_string('La cadena ingresada es: ' || cadena);
until(cadena = 'Salir');

menu := 2;
case (menu)
    value 1:
        print_string('La variable es 1');
        break
    value 2:
        print_string('La variable es 2');
        break
    value 3:
        print_string('La variable es 3');
        break
    [default: print_string('La variable no es 1, 2 o 3');]
end_case;

print_string('El programa ha finalizado.');
```

9.2 menu2.p

```
repeat
print_string('Menu:');
print_string('1. Potencia de un número');
print_string('2. Módulo de un número');
print_string('3. Bucle inverso');
print_string('4. Salir');
print_string('Ingrese opcion:');
read(opcion);

if(opcion = 1)
then
    print_string('Introduzca base de la potencia');
    read(base);

    print_string('Introduzca exponente de la potencia');
    read(exponente);

    resultado := base^exponente;
    print(resultado);
else
    if(opcion = 2)
    then
        print_string('Introduzca número');
        read(numero);

        print_string('Introduzca el módulo que desea realizar');
        read(modulo);

        resultado_modulo := numero%modulo;
        print(resultado_modulo);
    else
        if(opcion = 3)
        then
            inicio := 100;
            fin := 1;
            suma := 0;
            paso := 2;
            for i from inicio to fin step paso do
                suma := suma + i;
                print(suma);
            end_for;
            print_string('Resultado final: ');
            print(suma);
        else
            if(opcion = 4)
            then
                print_string('Saliendo del programa...');
```

```
            else
                print_string('Opcion invalida.');
            end_if;
        end_if;
    end_if;
end_if;

until (opcion = 4);
```

9.3 menu2_error.p

```
repeat
print_string('Menu:');
print_string('1. Potencia de un número');
print_string('2. Módulo de un número')   #error
print_string('3. Bucle inverso');
print_string('4. Salir');
print_string('Ingrese opcion:');
read(opcion);

if(opcion 1)  #error
then
    print_string('Introduzca base de la potencia');
    read(base);

    print_string('Introduzca exponente de la potencia');
    read(exponente);

    resultado := base^exponente;
    print(resultado);
else
    if(opcion = 2)
    then
        print_string('Introduzca número');
        read(numero)    #error

        print_string('Introduzca el módulo que desea realizar');
        read(modulo);

        resultado_modulo := numero%modulo;
        print(resultado_modulo);
    else
        if(opcion = 3)
        then
            inicio := 100;
            fin := 1;
            suma := 0;
            paso := 2;
            for i inicio to fin step paso do
                suma := suma + i;
                print(suma);
            end_for;
            print_string('Resultado final: ');
            print(suma);
        else
            if(opcion = 4)
            then
                print_string('Saliendo del programa...');
```

```
            else
                print_string('Opcion invalida.');
            #error
        end_if;
    end_if;
end_if;

until (opcion = 4);
```

10. CONCLUSIONES

10.1 Reflexión sobre el trabajo realizado

Mediante la realización de este trabajo, hemos llegado a la conclusión de que para generar un analizador sintáctico es necesario tener en cuenta muchos factores tanto en la reglas de producción como en la generación del árbol AST para que este tenga la menor cantidad de errores posibles y maneje de manera correcta todas las posibles situaciones a las que se pueda enfrentar.

10.2 Puntos fuertes y débiles del intérprete

Al analizar nuestro trabajo, identificamos ciertos puntos débiles que requieren atención y mejora. Estos puntos incluyen:

- La función **read_string** no lee cadenas con salto de líneas. Se pueden utilizar concatenaciones para simular los saltos de líneas.
- El programa no distingue entre mayúsculas y minúsculas. Esto se ve reflejado a la hora de poner los tokens, que siempre han de ser en minúsculas.
- La función **switch** solamente funciona para valores numéricos
- No muestra bien la línea en la que se encuentra el error. Muestra la siguiente línea de código.

Como punto fuerte tenemos que el analizador sintáctico es capaz de manejar de manera correcta la mayoría de casos de prueba utilizados a través de los ficheros de extensión .p a modo de tests.

11. BIBLIOGRAFÍA

Fernández García, N. L. (n.d.). *Procesadores de lenguajes*. UCO Moodle. https://moodle.uco.es/m2223/pluginfile.php/46428/mod_resource/content/3/Guion-practicas.pdf