Intérprete de pseudocódigo en inglés

PIE: pseudocode interpreter in English

Universidad de Córdoba
Escuela Politécnica Superior de Córdoba

2019 - 2020

Grado de Ingeniería informática
Especialidad: Computación

Tercer curso
Segundo cuatrimestre

Procesadores de lenguaje

Trabajo realizado por:
-Antonio Gómez Giménez
-Rafael Hormigo Cabello

Córdoba, Campus de Rabanales a 03/06/2020

Índice:

1.Introducción

Hemos realizado un intérprete de pseudocódigo en inglés utilizando Flex como herramienta para crear un analizador léxico, también utilizamos GNU Bison y Yacc para generar el analizador sintáctico-semántico. Como lenguaje interno utilizamos C++.

Nuestro documento se divide en las siguiente partes:

- Lenguaje de pseudocódigo: explicación del lenguaje de pseudocódigo para el que vamos a crear el intérprete.
- Tabla de símbolos: en este apartado se mostrará y explicará la codificación de la tabla de símbolos del analizador.
- Análisis léxico: apartado en el que se desarrollará la explicación del analizador léxico creado.
- Análisis sintáctico: apartado en el que se explica el analizador sintáctico desarrollado.
- AST: descripción del árbol AST creado.
- Funciones auxiliares: apartado donde se incluyen las funciones auxiliares para el código.
- Modo de obtención del intérprete: explicación de que contienen los directorios y los ficheros.
- Modo de ejecución: este apartado trata de los dos modos de introducir información al intérprete, de forma interactiva o desde un fichero.
- Ejemplos: se presentan y explican varios ejemplos propuestos.
- Conclusiones: se enumerarán y redactarán nuestras conclusiones sobre el trabajo realizado.

2.Lenguaje de pseudocódigo

En este apartado se van a comentar todos los componentes léxicos y sentencias creadas:

- **Componentes léxicos o tokens**:
    - Palabras reservadas. Para nuestro intérprete hemos añadido las siguiente palabras reservadas: *modulo, quotient, or, and, not, read, readstring, write, writestring, if, then, else, endif, while, do, endwhile repeat, until, for, from, to, step, endfor, clear, place, true* y *false*.

      Cabe destacar que no se distinguirá entre mayúsculas ni minúsculas y las palabras reservadas no se podrán utilizar como identificadores, para ello, en el caso de no diferenciar la mayúsculas y las minúsculas.

    - Identificadores. Los identificadores están compuestos por una serie de letras, dígitos y el subrayado, comienzan por una letra y no acaban con el símbolo de subrayado, ni tienen dos subrayados seguidos. Aparte no se distingue entre mayúsculas ni minúsculas.
      Ejemplos de identificadores válidos:
      -Identificadores válidos: ejemplo, ejemplo_1, ejemplo_1_a
      -Identificadores no válidos: _ejemplo, ejemplo_, ejemplo__1

    - Números. Se utilizan números enteros, reales de punto fijo y reales con notación científica y todos ellos son tratados conjuntamente como números. Unos ejemplos de números correctos son:
      -Números válidos: 23, 23.4, 23.3E4
      -Números no válidos: 23,25, 23.E

    - Cadena. Está compuesta por una serie de caracteres delimitados por comillas simples, permite la inclusión de la comilla simple utilizando la barra (\), las comillas exteriores no se almacenan como parte de la cadena y al escribir una cadena, se interpretan los caracteres especiales \n y \t para provocar un salto de línea o un salto de tabulador, respectivamente.

      Respecto al \n y \t, se controla cuando se realiza la visualización en pantalla de la cadena, esto se verá cuando se trate el WriteString.

      Un ejemplo de cadena sería el siguiente:
      -‘Prueba de cadena con salto de línea \n, tabulador \t y uso de las \’ comillas \’ simples’
      -‘Prueba de cadena’

- Operador de asignación. El operador de asignación está asociado a el símbolo :=, por ejemplo:
  Identificador := 43.4;
  Identificador := 'patata';
  Identificador := TRUE;
  Identificador := PI;
  Cabe destacar que la asignación puede hacerse concatenada, es decir asignar dos variables a la vez:
  identificador1 := identificador2 := 5;

- Operadores aritméticos. Son la suma, asociado al símbolo + (unario y binario), la resta, asociado al símbolo - (unario y binario), el producto, asociado al símbolo *, la división, asociado al símbolo /, la división entera, asociado al símbolo quotient, el módulo, asociado al símbolo modulo y la potencia, asociado al símbolo **. Unos ejemplos de uso serían:
  Identificador := 1 + 1;
  Identificador := 1 - 1;
  Identificador := 1 / 1;
  Identificador := 1 * 1;
  Identificador := 1 quotient 1;
  Identificador := 1 modulo 1;
  Identificador := 1 **1;

- Operador alfanumérico. El operador de alfanumérico está asociado a el símbolo ||, por ejemplo:
  b := 'cielos';
  Identificador := 'rasca' || b;

- Operadores relacionales de números y cadenas. Son menor que, asociado al símbolo < , menor o igual que, asociado al símbolo <=, mayor que, asociado al símbolo >, mayor o igual que, asociado al símbolo >=, igualdad, asociado al símbolo = y distinto, asociado al símbolo <>. Se controla que las comparaciones sean entre el mismo tipo, ya sea cadenas con cadenas y número con números por ejemplo. Se pueden usar identificadores siempre y cuando estos sean del mismo tipo que la comparación. Unos ejemplos de uso serían:

```
Identificador := 1 < 1;
a:=5;
if(Identificador<=false)then
        while(a>20)do
                a=a+1;
                b:=1>=a;
        endwhile
endif
```

```
repeat
        a:=5
until(a<>5)
Identificador := 1 = 1;
```

- Operadores de igualdad y desigualdad de valores lógicos. Funcionan igual que los explicados anteriormente y son igual, asociado al símbolo = y distinto, asociado al símbolo <>.

- Operadores lógicos. Son disyunción lógica, símbolo or, conjunción lógica, símbolo and y negación lógica, símbolo not. Un ejemplo sería:

```
for A from -1 to 0 do
        control := 'diferente'
        if ((A <= 0) and not (control <> 'stop')) then
                write(A);
                writestring(control);
        endif
endfor;
```

- Comentarios. Pueden ser de varias líneas delimitados por el símbolo # o de una línea donde todo lo que siga al carácter @ hasta el final de la línea es el comentario. Ejemplos de comentario en línea y comentario en párrafo:

```
# ejemplo
de comentario
de cuatro
líneas #
@ ejemplo de comentario que ocupa una línea
```

- Punto y coma. Se utiliza para indicar el fin de una sentencia y está representado por el símbolo ";". Un ejemplo de su uso es el siguiente:

```
readstring(B);
```

- **Sentencias**
  - Asignación. Se recalca como se debe realizar la asignación:
    -**identificador := expresión numérica;** donde se declara a identificador como una variable numérica y le asigna el valor de la expresión numérica. Las expresiones numéricas se formarán con números, variables numéricas y operadores numéricos
    -**identificador := expresión alfanumérica;** donde se 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 (||).

-**identificador := expresión lógica;** donde se declara a identificador como una variable lógica y le asigna el valor de la expresión lógica Las expresiones lógicas se formarán con expresiones relacionales o lógicas y variables lógicas.

- Lectura. Puede ser de dos tipos, dependera si es un string u otro tipo de variable:
  -**read (identificador);** donde se declara a identificador como variable numérica y le asigna el número leído.
  -**readstring (identificador);** donde se declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).

- Escritura. Puede ser de dos tipos, dependera si es un string u otro tipo de variable:
  -**write (expresión numérica);** donde el valor de la expresión numérica es escrito en la pantalla.
  -**writestring (expresión alfanumérica);** donde la cadena (sin comillas exteriores) es escrita en la pantalla y se permite la interpretación de comandos de saltos de línea (\n) y tabuladores (\t) que aparecen en la expresión alfanumérica. Por ejemplo:
  writestring('Introduzca el dato \n\t ->');

- Sentencias de control. Hay los siguiente tipos:

  -**Sentencia condicional simple.** Su estructura es:

```
if condición
        then lista de sentencias
endif
```

  -**Sentencia condicional compuesta.** Su estructura es:

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

  -**Bucle "mientras".** Su estructura es:

```
while condición do
        lista de sentencias
endwhile
```

-**Bucle "repetir".** Su estructura es:

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

-**Bucle "para".** Su estructura es:

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

*El paso "*step*" es opcional; en su defecto, tomará el valor 1. En este caso no es necesario usar la sentencia de bloque delimitada por llaves "{" y "}", ya que no es necesaria.

-Comandos especiales. Tenemos dos comandos especiales:
-**clear.** 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.

- Nuevos operadores. No hemos añadido ningún nuevo operador a nuestro intérprete.

- Observaciones. Cabe destacar que permitimos que una variable pueda cambiar de tipo durante la ejecución del intérprete, pudiendo pasar de bool a numérico e incluso string.

3.Tabla de símbolos

Para la tabla de símbolos hemos utilizado la clase *table* que tiene tanto .hpp como .cpp y también se ha usado la clase *symbol* ya que *tableInterface* usa esta clase que también tiene tanto .hpp como .cpp. Por tanto se van a explicar estas clases:

table: esta clase es la tabla donde se guardarán los símbolos usados para el intérprete, hereda de tableInterface, esta clase se usa para virtualizar los métodos que esta clase tendrá que tener.

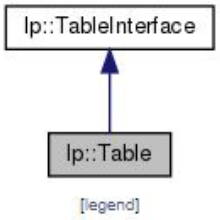


Figura 3.1

Los métodos que tiene esta clase son los siguientes:

```
bool lp::Table::lookupSymbol(const std::string & name) const
{
    if (this->_table.find(name) !=  this->_table.end())
        return true;
    else
        return false;
}


lp::Symbol * lp::Table::getSymbol(const std::string & name)
{
 #ifndef NDEBUG
   // Precondition
   assert (this->lookupSymbol(name) == true);
 #endif //NDEBUG

    return  this->_table[name];
}



void lp::Table::installSymbol(Symbol * s)
{
 #ifndef NDEBUG
  // Precondition
   assert (this->lookupSymbol(s->getName()) == false);
 #endif //NDEBUG

   // The pointer to symbol is inserted in the map
    this->_table[s->getName()] = s;

 #ifndef NDEBUG
  // Postcondition
   assert (this->lookupSymbol(s->getName()) == true);
 #endif //NDEBUG
}
```

Figura 3.2

```
void lp::Table::eraseSymbol(const std::string & name)
{
 #ifndef NDEBUG
  // Precondition
   assert (this->lookupSymbol(name) == true);
 #endif //NDEBUG

   // The symbol "name" is deleted from the map
    this->_table.erase(name);

 #ifndef NDEBUG
  // Postcondition
   assert (this->lookupSymbol(name) == false);
 #endif //NDEBUG
}


void lp::Table::printTable()
{
    for(std::map<std::string, lp::Symbol *>::const_iterator it = this->_table.begin();
        it != this->_table.end();
        ++it)
    {
        std::cout<<it->first<<", "<<getSymbol(it->first)->getToken()<< std::endl;
    }
}
```

Figura 3.3

-El método *lookupSymbol* nos permite ver si hay un símbolo en la tabla con el nombre pasado al método.
-El método *getSymbol* nos devuelve el símbolo pedido.
-El método *installSymbol* nos permite instalar un nuevo símbolo en la tabla de símbolos.
-El método *eraseSymbol* nos permite borrar un símbolo de la tabla de símbolos.
-El método *printTable* nos permite imprimir la tabla con sus respectivos símbolos.

symbol: esta clase son los símbolos que usará nuestro intérprete que se almacenarán en nuestra tabla creada por la clase *table*, hereda de *symbolInterface*, esta clase se usa para virtualizar los métodos que esta clase tendrá que tener.

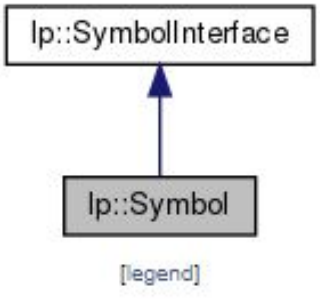


Figura 3.4

Los métodos que tiene esta clase son los siguientes:

```
inline const std::string & getName() const
{
    return this->_name;
}



\brief   Public method that returns the token of the Symbol
\note    Inline function
\pre     None
\post    None
\return  The token of the Symbol
\sa      getName, getType

inline int getToken() const
{
    return this->_token;
}



\name Modifiers



\brief   This functions modifies the name of the Symbol
\note    Inline function
\param   name: new name of the Symbol
\pre     None
\post    The name of the Symbol is equal to the parameter
\return  void
\sa      setToken

inline void setName(const std::string & name)
{
    this->_name = name;
}



\brief   This functions modifies the token of the Symbol
\note    Inline function
\param   token: new value of the Variable
\pre     None
\post    The token of the Symbol is equal to the parameter
\return  void
\sa      setName

inline void setToken(int token)
{
    this->_token = token;
}
```

Figura 3.5

En la figura 3.5 podemos ver los getter y los setters de la clase donde se permite tratar tanto con el nombre del símbolo como el token asociado al mismo.

También tiene los siguiente métodos:

```
bool lp::Symbol::operator==(const lp::Symbol & s) const
    {
        return ( this->getName() == s.getName() );
    }


bool lp::Symbol::operator <(const lp::Symbol & s) const
{
   if ( this->getName() < s.getName() )
        return true;
  else
      return false;
}
```

Figura 3.6

Aquí vemos que se han creado dos operadores que nos permiten comparar el nombre de dos símbolos y ver si el nombre de un símbolo es menor que otro.

Como breve resumen, se puede decir que, se crea la clase *symbol* para representar los símbolos donde almacenamos el token y el nombre y se crea la clase *table* para almacenar estos símbolos y por ahí empezar a trabajar una vez tenemos la tabla con sus respectivos símbolos.

Cabe destacar que de symbol heredan cuatro clases que son las siguientes:

builtin: de esta clase a su vez heredan tres clases, estas son builtinParameter0, builtinParameter1 y builtinParameter2. El objetivo de estas tres clases es similar, construir un nuevo símbolo a partir del nombre, token y los parámetros que serán 0, 1 o 2.

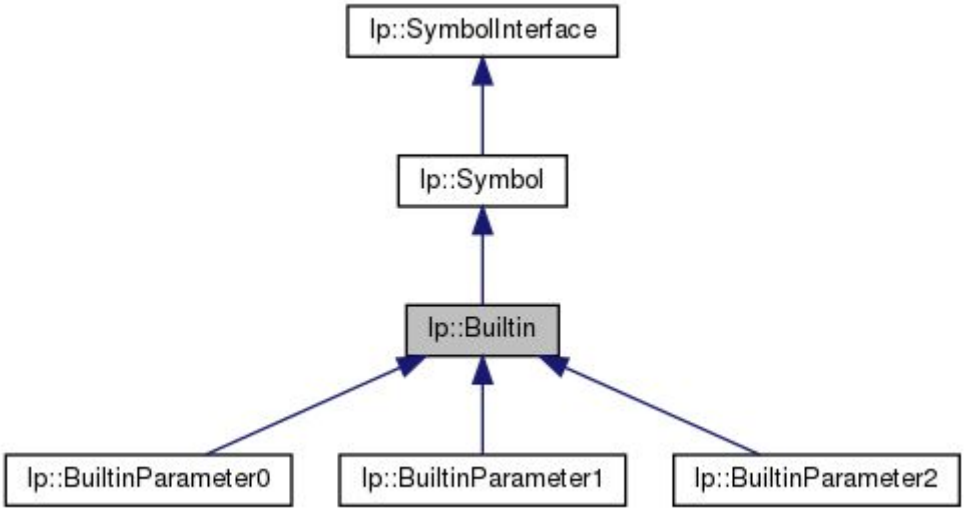


Figura 3.7

constant: de esta clase a su vez se heredan dos clases, estas son LogicalConstant y NumericConstant. Estas clases se crean para crear variables Constantes de tipo Logical y Numeric. Esta variables no podrán ser modificadas porque serán constantes.

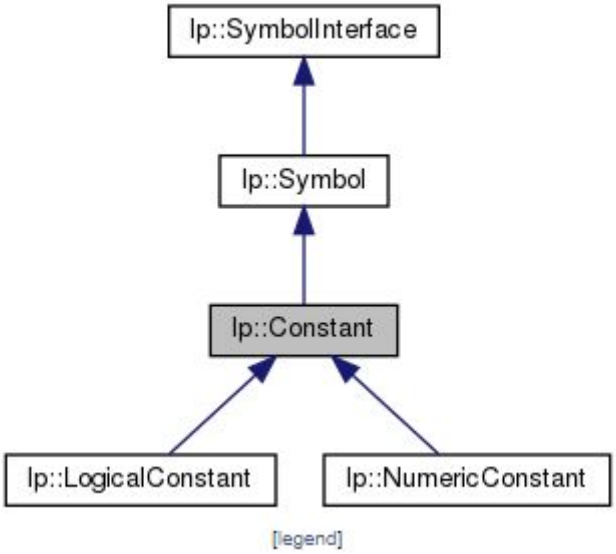


Figura 3.8

keyword: esta clase permite crear palabras clave, estas se utilizarán en la clase init para así poder crear las palabras reservadas.

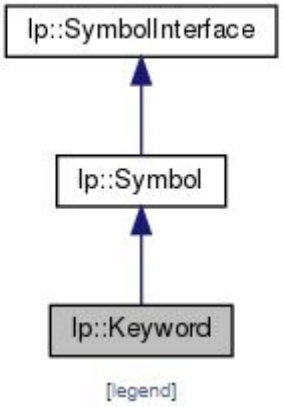


Figura 3.9

variable: de esta clase a su vez se heredan tres clases, estas son LogicalVariable, NumericVariable y StringVariable. Estas clases se crean para crear definir el tipo de una variable de esta forma las variables serán de tipo Bool, de tipo numérico y de tipo String.

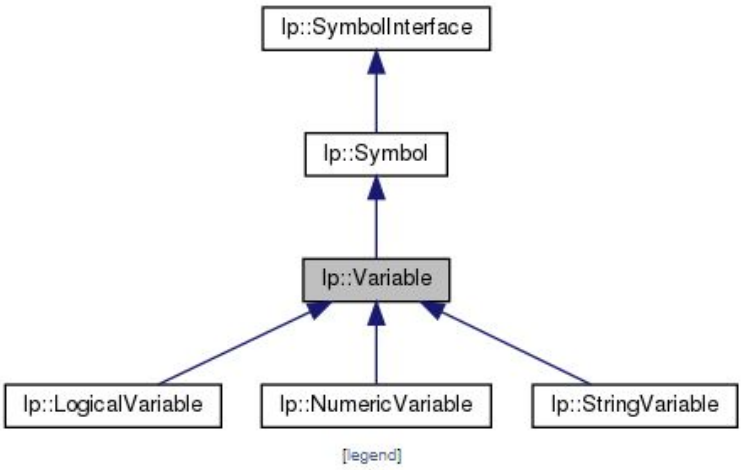


Figura 3.10

Cabe destacar que también se crea la clase init. Esta clase permite inicializar la tabla con los valores que queramos, en este caso serán las palabra clave y aquellas que sean constantes. Esto se consigue ya que al crear una tabla se crea la clase init donde ya se encuentra en el vector keywords la palabras clave y se instalan. Se ve más claro en la imagen de ejemplo:

```
static struct {
          std::string name ;
          int token;
          } keyword[] = {
                        "read",  READ,
                    "readstring", READSTRING,
                    "write", WRITE,
                    "writestring", WRITESTRING,
                              "if",    IF,        // NEW in example 17
                    "then", THEN,
                              "else",  ELSE,      // NEW in example 17
                    "endif", ENDIF,
                              "while",  WHILE,    // NEW in example 17
                    "do", DO,
                    "endwhile", ENDWHILE,
                    "repeat", REPEAT,
                    "until", UNTIL,
                    "for", FOR,
                    "from", FROM,
                    "to", TO,
                    "step", STEP,
                    "endfor", ENDFOR,
                    "clear", CLEAR,
                    "place", PLACE,
                    "not", NOT,
                    "modulo", MODULO,
                    "or", OR,
                    "and", AND,
                    "quotient", QUOTIENT,
                      "",      0
                     };
```

Figura 3.11

Por esto es necesario crear todas las clases anteriores ya que son necesarias como la clase keyWord o las clases buildingParameter0 o buildingParameter2.

4.Análisis léxico

Para realizar el análisis léxico hemos modificado el fichero interpreter.l. Para ello, voy a mostrar los cambios realizados, es decir, las nuevas expresiones regulares usadas y las nuevas reglas que hemos implementado.

Expresiones regulares:

- NUMBER1 y NUMBER2. Estas expresiones regulares son los número que nuestro intérprete considerará como números y posteriormente trabajaremos con ellos.

```
NUMBER1 {DIGIT}+(\.{DIGIT}+)?

NUMBER2 {DIGIT}(\.{DIGIT}+)?(E[+-]?{DIGIT}+)?
```

Figura 4.1

- BADNUMBER. Esta expresión ha sido añadida por nosotros para controlar en el léxico, aquellos números que nosotros no consideramos válidos para nuestro intérprete.

```
BADNUMBER {DIGIT}+(\.)(E[+-]?{DIGIT}+)?|{DIGIT}+(\.{DIGIT}+)?(E[+-]?)({DIGIT}+)?|{DIGIT}+(\.)(E[+-]?)
```

Figura 4.2

- IDENTIFIER. Esta expresión regular nos da los identificadores que nosotros consideramos correctos y que posteriormente trabajamos con ellos.

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

Figura 4.3

- BADIDENTIFIER. Esta expresión ha sido añadida por nosotros para controlar en el léxico, recoge aquellos identificadores que nosotros no consideramos válidos para nuestro intérprete, para su posterior tratamiento.

```
BADIDENTIFIER ({LETTER}|\_|{DIGIT})({LETTER}+|{DIGIT}+|(\_)+)+({LETTER}|{DIGIT}|\_)?
```

Figura 4.4

- COM_LINE. Esta expresión reconoce los comentarios en línea que son aquellos que empiezan por @.

```
COM_LINE            @(.)*$
```

Figura 4.5

Reglas usadas:

- Cadenas. Se ha realizado una regla basándose en estados para recoger las cadenas para nuestro programa. El conjunto de estados es el siguiente:

```
%x ESTADO_CADENA
```

Figura 4.6

Siendo los estados:

```
"'"                     {
                         /* Se activa el reconocimiento de la cadena */
                          BEGIN ESTADO_CADENA;
                        }
<ESTADO_CADENA>"'"      { /* Fin de la cadena: se vuelve al estado por defecto */
                          BEGIN 0;
                          yytext[yyleng-1]='\0';
                          yylval.string = yytext;
                          return STRING;
                        }
<ESTADO_CADENA>"\\\'"   {yymore();}
<ESTADO_CADENA>.        {yymore();}
```

Figura 4.7

Lo que se realiza en este caso es que si empareja la “‘“, entonces se reconoce la cadena. Dependiendo de si es una letra, dígito o una barra con una “‘“, entonces se añade a la cadena. Por último, si vuelve a emparejar con “‘“, entonces termina la cadena.

- Comentarios. Los comentarios también se han realizado por el método anterior siendo el estado Q1:

```
%x ESTADO_CADENA Q1
```

Figura 4.8

Por tanto lo que hemos hecho ha sido lo siguiente:

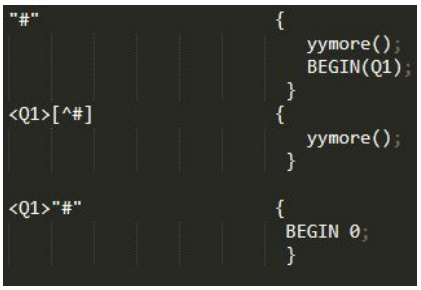


Figura 4.9

Si empareja la “#” entonces comienza el estado Q1 y se reconoce como un comentario. Mientras que no se vuelva a encontrar “#” se añade al comentario y cuando se encuentra se termina el comentario.

Esto es para los comentario que no son en línea, si es un comentario en línea se hace lo siguiente:

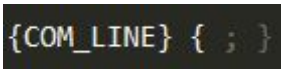


Figura 4.10

Esto principalmente hace que el intérprete reconozca el comentario en línea y no haga nada al respecto.

- Números. Para los números tenemos dos tipos de reglas, para los números correctos y para los números incorrectos. Para los números correctos es la siguiente regla:

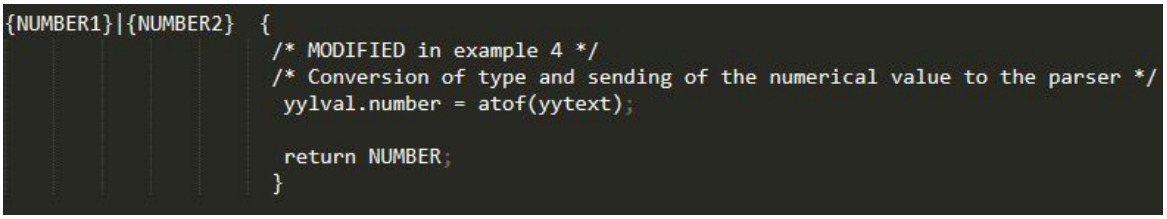


Figura 4.11

En esta regla se coge el número y se pasa a float para posteriormente pasarlo al sintáctico.

Respecto a los números incorrectos se ha utilizado la siguiente regla:

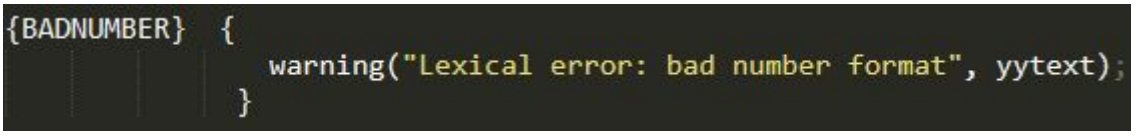


Figura 4.12

Esta regla nos avisa de que el número introducido no está en el formato correcto.

- Identificadores. Igual que para los números, tenemos dos tipos de reglas, una para los identificadores correctos y otra para los identificadores incorrectos. Para los identificadores correctos tenemos la siguiente regla:

```
{IDENTIFIER}        {
                        /* NEW in example 7 */
                        std::string identifier(yytext);
            for (int i = 0; i < identifier.size(); i++) {
              identifier[i] = tolower(identifier[i]);
            }

                      /*
                        strdup() function returns a pointer to a new string
                        which  is a duplicate of the string yytext
                      */
                        yylval.identifier =  strdup(identifier.c_str());


                      /* 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;
                          }

                      /* MODIFIED in example 11 */
                      /*
                          If the identifier is in the table of symbols then its token is returned
                              The identifier can be a variable or a numeric constant
                      */
                      else
                          {
                              lp::Symbol  *s = table.getSymbol(identifier);

/*
                              std::cout << "lex: "<< s->getName()
                                        << "token " << s->getToken()
                                        << std::endl;
*/

                              /*  If the identifier is in the table then its token is returned  */
                              return s->getToken();
                          }
}
```

Figura 4.13

En esta regla, primero transformamos el identificador que ha entrado a mayúscula ya que nuestro intérprete necesita que esté todo en minúscula para que no haya diferencias entre mayúscula y minúscula.
Posteriormente le pasamos este identificador al sintáctico y comprobamos si este identificador no está en la tabla de símbolos. Si no está, se añade como indefinido y con valor 0.0, una vez se haga la asignación ya se verá al tipo que pertenece. Si ya se encuentra en la tabla de símbolos, significa que o es una constante o ya se encontraba definida, por tanto se devuelve el token del símbolo encontrado.

En el caso de que sea un identificador no válido se realiza la siguiente regla:

```
{BADIDENTIFIER} {
                  warning("Lexical error: bad identifier", yytext);
                }
```

Figura 4.14

En esta regla, simplemente se le avisa al usuario de que el identificador introducido no es válido.

- Resto de reglas. El resto de reglas ya estaban creadas o son muy sencillas, en este caso voy a mostrar las siguientes:

```
"-"     { return MINUS;  }          /* NEW in example 3 */
"+"     { return PLUS;   }          /* NEW in example 3 */

"*"     { return MULTIPLICATION; }  /* NEW in example 3 */
"/"     { return DIVISION; }        /* NEW in example 3 */

"("     { return LPAREN; }          /* NEW in example 3 */
")"     { return RPAREN; }          /* NEW in example 3 */


"**"        { return POWER; }           /* NEW in example 5 */

"=" { return EQUAL; }           /* NEW in example 15 */

":="    { return ASSIGNMENT; }      /* NEW in example 7 */

"<>"    { return NOT_EQUAL; }       /* NEW in example 15 */

">="    { return GREATER_OR_EQUAL; }/* NEW in example 15 */

"<="    { return LESS_OR_EQUAL; }   /* NEW in example 15 */

">"     { return GREATER_THAN; }    /* NEW in example 15 */

"<"     { return LESS_THAN; }       /* NEW in example 15 */

"||"  { return CONCAT; } /* NEW in final work */
```

Figura 4.15

Como podemos observar, simplemente si empareja una regla, se devuelve el token para que empareje con la tabla de símbolos.
Por tanto, si quisiéramos cambiar el símbolo + por suma, simplemente habría que cambiarlo en el léxico y el resto del programa se mantendría igual ya que dentro de la tabla se llama PLUS y esto no varía.

5.Análisis sintáctico:

- **Símbolos de la gramática:**
  - Símbolos terminales:
    - VARIABLE: representación de las variables.
    - NUMBER: representación de los números.
    - STRING: representación de las cadenas.
    - BOOL: representación de valores lógicos.
    - CONSTANT: representación de constantes.
    - SEMICOLON: símbolo final de línea ";".
    - RPAREN: paréntesis derecho.
    - LPAREN: paréntesis izquierdo.
    - COMMA: coma ",".
    - Símbolos no terminales de sentencias de control: READ, READSTRING, WRITE, WRITESTRING, IF, THEN, ELSE, ENDIF, WHILE, DO, ENDWHILE, REPEAT, UNTIL, FOR, FROM, TO, STEP, ENDFOR, CLEAR y PLACE.
    - Símbolos no terminales de operadores: PLUS (+), MINUS (-), MULTIPLICATION (*), DIVISION (/), POWER (**), QUOTIENT, MODULO, LESS_THAN (<), GREATER_THAN (>), GREATER_OR_EQUAL (<=), LESS_OR_EQUAL (>=), EQUAL (=), NOT_EQUAL (<>), ASSIGNMENT (:=), CONCAT (||), OR y AND.
    - ControlSymbol: símbolo de control para el modo interactivo.
    - error: símbolo terminal para indicar errores.

  - Símbolos no terminales:
    - Program: Símbolo inicial.
    - Stmt: símbolo de sentencia.
    - StmtList: símbolo de lista de sentencias.
    - exp: símbolo de expresión.
    - Símbolos no terminales de sentencias de control: read, readstring, write, writestring, if, while, repeat, for, clear y place.
    - cond: símbolo de condición.
    - asgn: símbolo no terminal de la asignación.
    - listOfExp: lista de expresiones.
    - restOfListOfExp: resto de la lista de expresiones.

- **Reglas de producción de la gramática:**

  p {

  program -> stmtlist
  stmtlist -> ε
  stmtlist -> stmtlist stmt
  stmtlist -> stmtlist error

stmt -> SEMICOLON
stmt -> asgn SEMICOLON
stmt -> write SEMICOLON
stmt -> read SEMICOLON
stmt -> writestring SEMICOLON
stmt -> readstring SEMICOLON
stmt -> if SEMICOLON
stmt -> while SEMICOLON
stmt -> repeat SEMICOLON
stmt -> for SEMICOLON
stmt -> place SEMICOLON
stmt -> clear SEMICOLON
controlSymbol -> ε
write -> WRITE LPAREN exp RPAREN
read -> READ LPAREN VARIABLE RPAREN
reads -> READ LPAREN CONSTANT RPAREN (para errores)
writestring -> WRITESTRING LPAREN exp RPAREN
readstring -> READSTRING LPAREN VARIABLE RPAREN
readstring -> READSTRING LPAREN CONSTANT RPAREN (para errores)
if -> IF controlSymbol LPAREN CONSTANT RPAREN THEN stmtlist ENDIF (para errores)
if -> IF controlSymbol LPAREN CONSTANT RPAREN THEN stmtlist ELSE stmtlist ENDIF (para errores)
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 controlSymbol stmtlist UNTIL cond
for -> FOR controlSymbol VARIABLE FROM exp TO exp STEP exp DO stmtlist ENDFOR
for -> FOR controlSymbol VARIABLE FROM exp TO exp DO stmtlist ENDFOR
cond -> LPAREN exp RPAREN
asgn -> VARIABLE ASSIGNMENT exp
asgn -> VARIABLE ASSIGNMENT asgn
asgn -> CONSTANT ASSIGNMENT exp (para errores)
asgn -> CONSTANT ASSIGNMENT asgn (para errores)
clear -> CLEAR
place -> PLACE LPAREN exp COMMA exp RPAREN
exp -> STRING
exp -> exp CONCAT exp
exp -> NUMBER
exp -> exp PLUS exp
exp -> exp MINUS exp

exp -> exp MULTIPLICATION exp
exp -> exp DIVISION exp
exp -> exp QUOTIENT exp
exp -> LPAREN exp RPAREN
exp -> PLUS exp %prec UNARY
exp -> MINUS exp %prec UNARY
exp -> exp MODULO exp
exp -> exp POWER exp
exp -> VARIABLE
exp -> CONSTANT
exp -> BUILTIN LPAREN listOfExp RPAREN
exp -> exp GREATER_THAN exp
exp -> exp GREATER_OR_EQUAL exp
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 -> ε
listOfExp -> exp restOfListOfExp
restOfListOfExp -> ε
restOfListOfExp -> COMMA exp restOfListOfExp

}

- **Acciones semánticas:**
  - write -> WRITE LPAREN exp RPAREN
    Se crea un nuevo nodo WriteStmt con la expresión “exp”. Dentro se comprobará si es number o bool o si es otra cosa y dará error.
  - read -> READ LPAREN VARIABLE RPAREN
    Se crea un nuevo nodo ReadStmt pasándole la variable, dentro de la clase ReadStmt se comprobará que la variable sea numérica.
  - read -> READ LPAREN CONSTANT RPAREN
    Si se intenta leer una constante por teclado se dará un error
  - writestring -> WRITESTRING LAPREN exp RPAREN
    Se crea un nuevo nodo WriteStringStmt y se le pasa pa expresión, dentro de la clase se comprobará si la expresión es una cadena o no.
  - readstring -> READSTRING LPAREN VARIABLE RPAREN
    Se crea un nodo de tipo ReadStmt pasándole la expresión que se comprobará dentro de la clase para comprobar el tipo.
  - if -> IF controlSymbol LPAREN CONSTANT RPAREN THEN stmtlist ENDIF
    Si la condición es constante se produce un warning advirtiendo de que el if es inútil, aunque aún así se continúa con la ejecución.

- if -> IF controlSymbol LPAREN CONSTANT RPAREN THEN stmtlist ELSE stmtlist ENDIF
  Mismo caso que antes pero para la versión con else.
- if -> IF controlSymbol cond THEN stmtlist ENDIF
  Se crea un nuevo nodo ifStmt con la condición y la lista de sentencias. Dentro de la clase se implementa como un if de C++.
- IF controlSymbol cond THEN stmtlist ELSE stmtlist ENDIF
  Mismo caso que el anterior pero para el caso con else.
- while -> WHILE controlSymbol cond DO stmtlist ENDWHILE
  Creamos un nodo WhileStmt con la condición y la lista de sentencias. En la clase se comprueba que la condición sea lógica y si lo es se ejecuta con un while de C++.
- repeat -> REPEAT controlSymbol stmtlist UNTIL cond
  Creamos un nodo RepeatStmt con la condición y la lista de sentencias. La condición se comprueba dentro de la clase y se ejecuta como un do while de C++ pero con la condición negada.
- for -> FOR controlSymbol VARIABLE FROM exp TO exp STEP exp DO stmtlist ENDFOR
  Se crea un nodo ForStmt con la variable y las tres expresiones, from será desde dónde va la variable, to hasta dónde irá y step será el incremento o decremento que sufrirá hasta llegar al final. Se ejecuta como un for de C++.
- for -> FOR controlSymbol VARIABLE FROM exp TO exp DO stmtlist ENDFOR
  En este caso se llama al constructor sin step, por lo que dentro de la clase se le asignará 1 al paso.

6.Código de AST

Dentro del código AST nos encontramos con dos clases principales ExpNode y Statement.

- ExpNode: ésta clase es la clase padre de todas las expresiones, de ella nacen los operadores, las variables y las constantes como se puede observar en la figura 3.1.

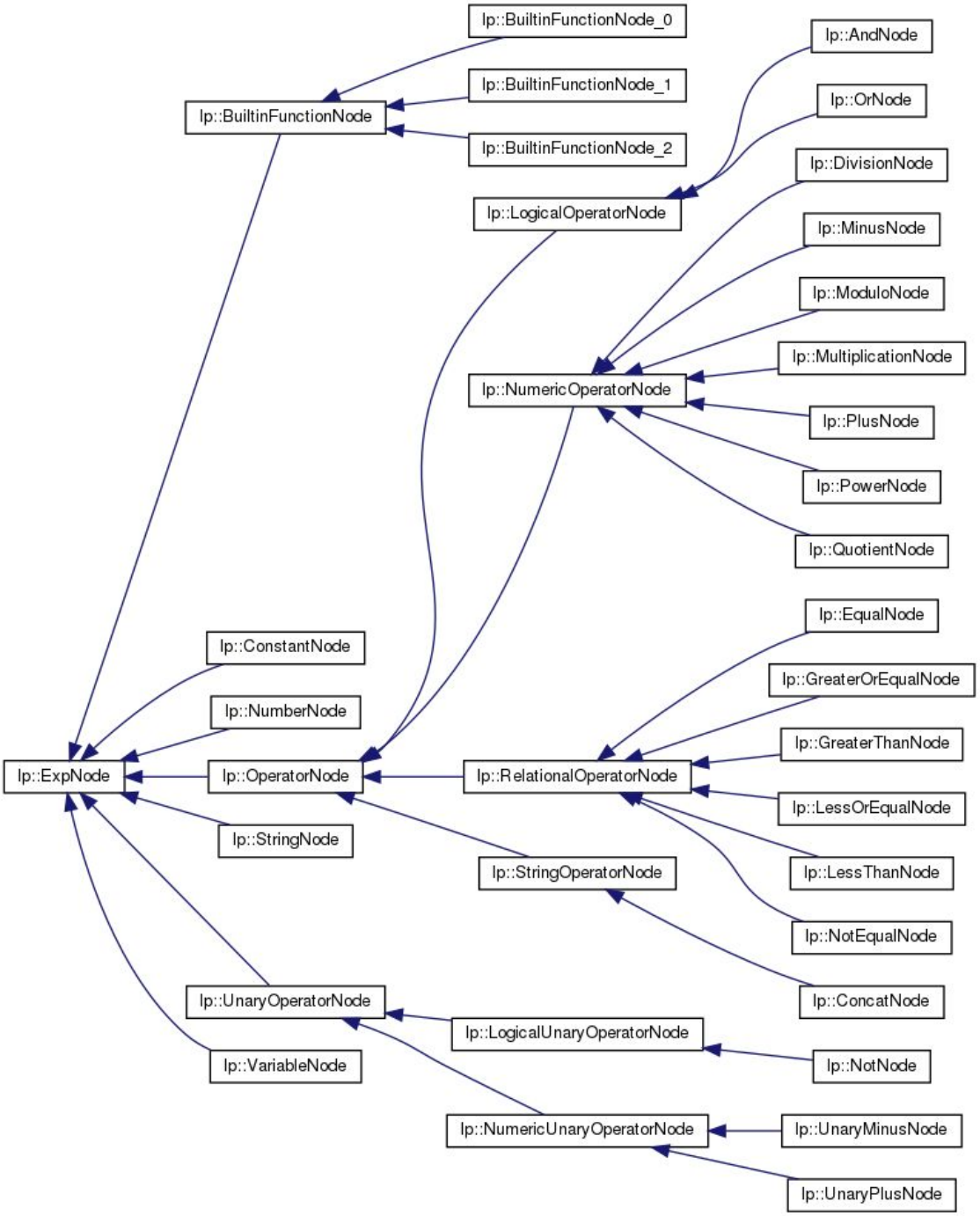


Figura 6.1

- Las clases BuiltInFunctionNode son esquemáticas y no tienen ninguna funcionalidad. Estas clases definen la estructura que tendrán las demás clases que sí serán útiles.

```
class BuiltinFunctionNode : public ExpNode
{
  protected:
    std::string _id; //!< Name of the BuiltinFunctionNode

  public:
/*!
    \brief Constructor of BuiltinFunctionNode
    \param id: string, name of the BuiltinFunction
    \post  A new BuiltinFunctionNode is created with the parameter
*/
  BuiltinFunctionNode(std::string id)
    {
        this->_id = id;
    }

};
```

Figura 6.2

La clase no numerada es la clase padre y solo define el constructor para una clase con un identificador.

```
class BuiltinFunctionNode_0 : public BuiltinFunctionNode
{
  public:
/*!
    \brief Constructor of BuiltinFunctionNode_0 uses BuiltinFunctionNode's constructor as member initializer
    \param id: string, name of the BuiltinFunction
    \post  A new BuiltinFunctionNode_2 is created with the parameter
*/
  BuiltinFunctionNode_0(std::string id): BuiltinFunctionNode(id)
    {
        //
    }

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



    /*!
        \brief   Print the BuiltinFunctionNode_0
        \return  void
        \sa      evaluate()
    */
      void print();

    /*!
        \brief   Evaluate the BuiltinFunctionNode_0
        \return  double
        \sa      print
    */
      double evaluateNumber();
};
```

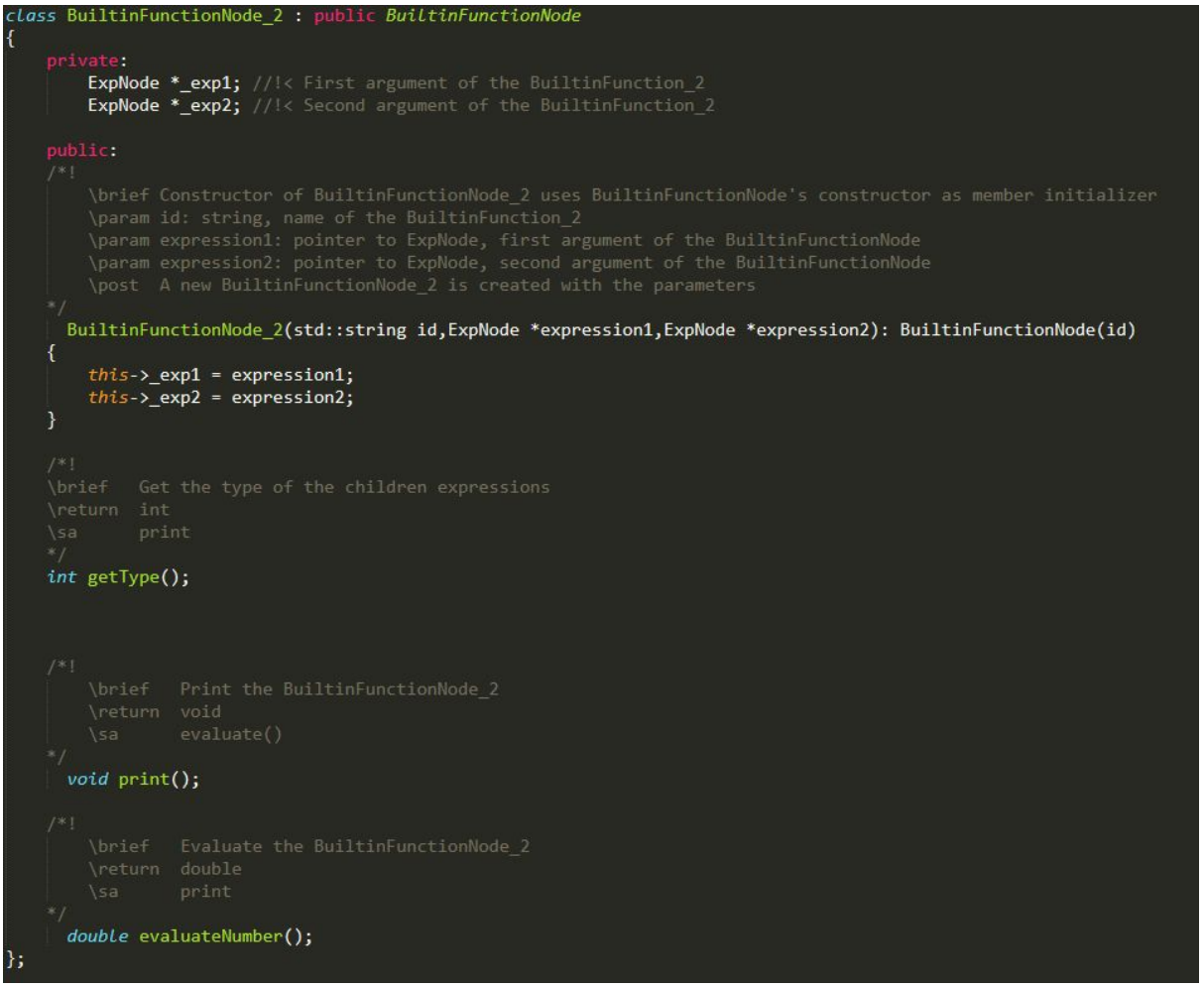
Figura 6.3

La clase número 0 corresponde a una clase con funciones getType(), devuelve el tipo del nodo (numérico, lógico, cadena o indefinido), la función print(), que imprime por consola la información del nodo y la función evaluateNumber(), que devuelve el nodo evaluado, en este caso devuelve un valor numérico.

```
class BuiltinFunctionNode_1: public BuiltinFunctionNode
{
  private:
    ExpNode *_exp;  //!< Argument of the BuiltinFunctionNode_1

  public:
/*!
    \brief Constructor of BuiltinFunctionNode_1 uses BuiltinFunctionNode's constructor as member initializer
    \param id: string, name of the BuiltinFunction
    \param expression: pointer to ExpNode, argument of the BuiltinFunctionNode_1
    \post  A new BuiltinFunctionNode_1 is created with the parameters
*/
  BuiltinFunctionNode_1(std::string id, ExpNode *expression): BuiltinFunctionNode(id)
    {
        this->_exp = expression;
    }

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

    /*!
        \brief   Print the BuiltinFunctionNode_1
        \return  void
        \sa      evaluate()
    */
      void print();

    /*!
        \brief   Evaluate the BuiltinFunctionNode_1
        \return  double
        \sa      print
    */
      double evaluateNumber();
};
```

Figura 6.4

La clase número 1 es la estructura para una clase con un nodo expresión dentro de ella además del identificador.



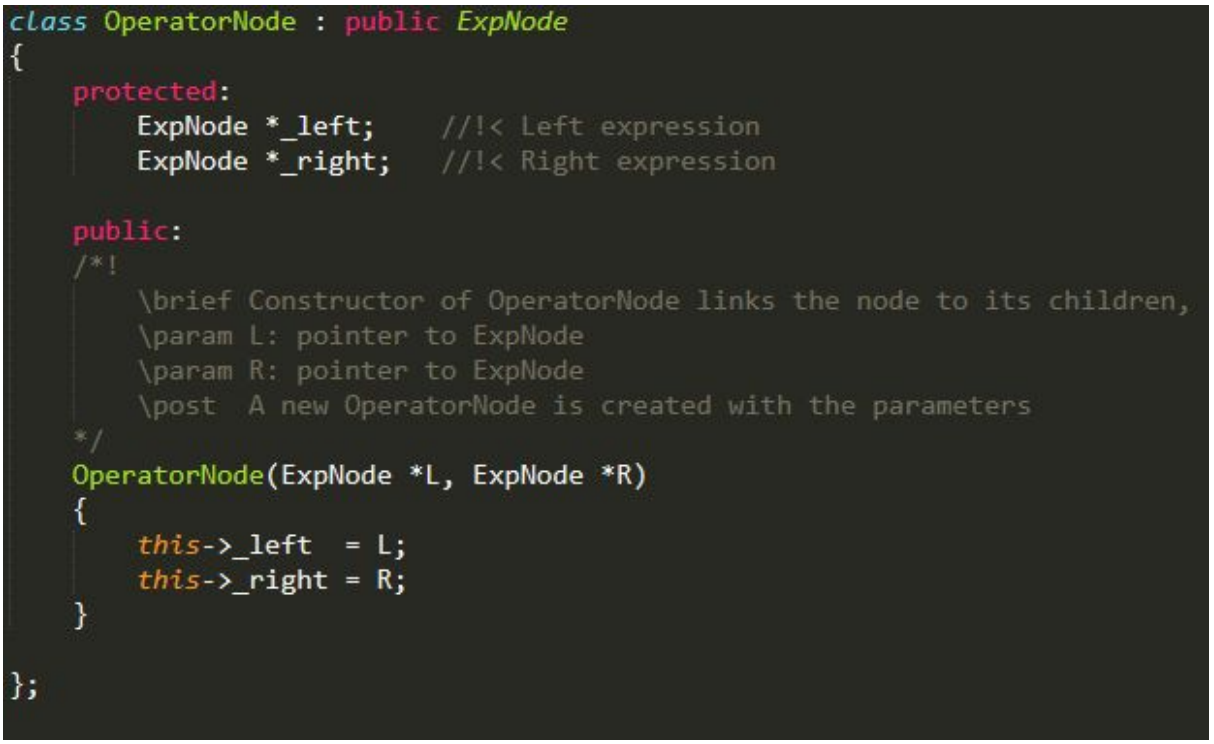
```
class BuiltinFunctionNode_2 : public BuiltinFunctionNode
{
    private:
        ExpNode *_exp1; //!< First argument of the BuiltinFunction_2
        ExpNode *_exp2; //!< Second argument of the BuiltinFunction_2

    public:
    /*!
        \brief Constructor of BuiltinFunctionNode_2 uses BuiltinFunctionNode's constructor as member initializer
        \param id: string, name of the BuiltinFunction_2
        \param expression1: pointer to ExpNode, first argument of the BuiltinFunctionNode
        \param expression2: pointer to ExpNode, second argument of the BuiltinFunctionNode
        \post  A new BuiltinFunctionNode_2 is created with the parameters
    */
      BuiltinFunctionNode_2(std::string id,ExpNode *expression1,ExpNode *expression2): BuiltinFunctionNode(id)
    {
        this->_exp1 = expression1;
        this->_exp2 = expression2;
    }

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



    /*!
        \brief   Print the BuiltinFunctionNode_2
        \return  void
        \sa      evaluate()
    */
      void print();

    /*!
        \brief   Evaluate the BuiltinFunctionNode_2
        \return  double
        \sa      print
    */
      double evaluateNumber();
};
```

Figura 6.5

La clase número dos marca la estructura para una clase con dos nodos expresión dentro de ella.

Como ya hemos comentado las demás clases siguen uno de estos patrones de estructura por lo que sólo añadiremos capturas de los cambios interesantes que veamos en ellas para no extender innecesariamente este documento.

- La clase OperatorNode es la clase padre de las clases de operadores numéricos, relacionales, lógicos y de cadenas.

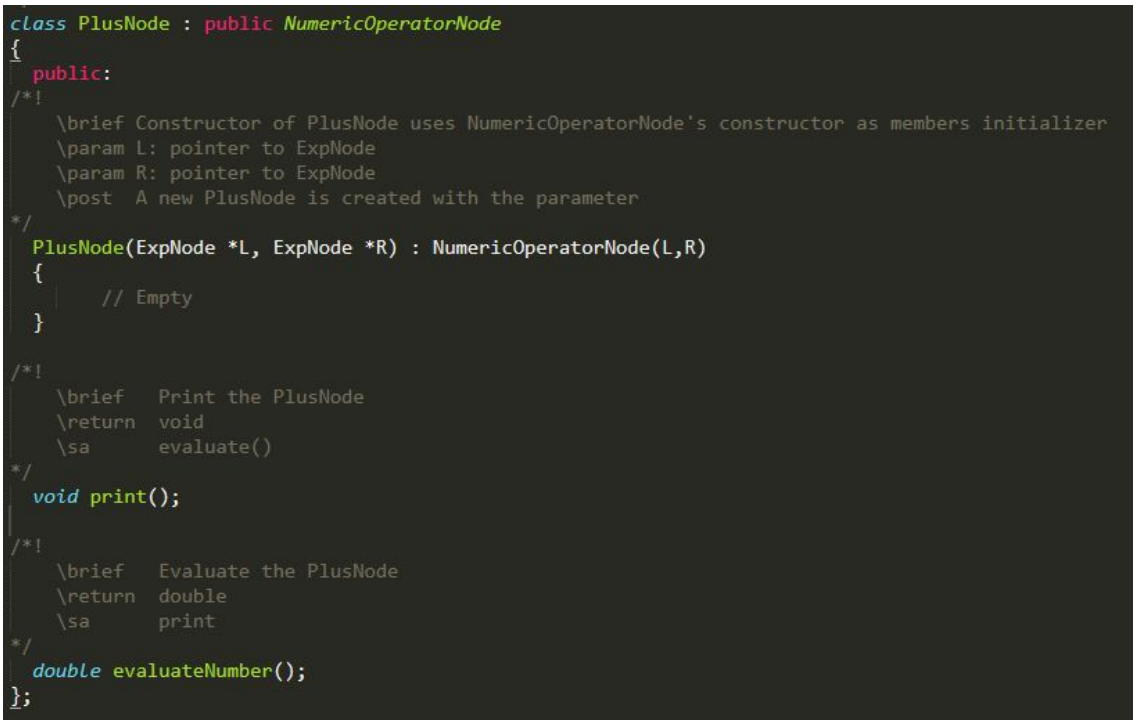


```
class OperatorNode : public ExpNode
{
    protected:
        ExpNode *_left;    //!< Left expression
        ExpNode *_right;   //!< Right expression

    public:
    /*!
        \brief Constructor of OperatorNode links the node to its children,
        \param L: pointer to ExpNode
        \param R: pointer to ExpNode
        \post  A new OperatorNode is created with the parameters
    */
    OperatorNode(ExpNode *L, ExpNode *R)
    {
        this->_left  = L;
        this->_right = R;
    }

};
```

Figura 6.6

Declara los dos nodos de expresión que componen cualquier operación binaria. Cada tipo de operador define también una función getType que devuelve el tipo de la operación (número, cadena o booleano).

Operadores numéricos: tenemos un total de 7 operadores numéricos (suma, resta, división, multiplicación, división entera, módulo y potencia), los cuales tienen la misma estructura todos. Por ejemplo el operador de suma:



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

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

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

Figura 6.7

Consta de 3 funciones, el constructor, que llama al constructor padre, print, que muestra información sobre el nodo, y por último evaluateNumber, que devuelve el valor numérico de aplicar la operación.

Operadores relacionales: operadores que actúan sobre cadenas, números y booleanos (igualdad, mayor, menor, mayor o igual, menor o igual y desigualdad). Tienen la misma estructura que los operadores numéricos solo que la función de evaluación devuelve un booleano. Ejemplo:

```
class EqualNode : public RelationalOperatorNode
{
  public:

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

/*!
    \brief   Evaluate the EqualNode
    \return  bool
    \sa      print()
*/
  bool evaluateBool();;
};
```

Figura 6.8

Dentro de la función de evaluación encontramos la comprobación de en qué caso estamos, si es cadena, número o booleano.

```
bool lp::EqualNode::evaluateBool()
{
    bool result = false;

    if (this->getType() == BOOL)
    {
        switch(this->_left->getType()){
            case NUMBER:
                double leftNumber, rightNumber;
                leftNumber = this->_left->evaluateNumber();
                rightNumber = this->_right->evaluateNumber();

                // ERROR_BOUND to control the precision of real numbers
                result = ( std::abs(leftNumber - rightNumber) < ERROR_BOUND );
            break;
            case BOOL:
                bool leftBoolean, rightBoolean;
                leftBoolean = this->_left->evaluateBool();
                rightBoolean = this->_right->evaluateBool();

                //
                result = (leftBoolean == rightBoolean);
                break;
            case STRING:
            {
                std::string leftString, rightString;
                leftString = this->_left->evaluateString();
                rightString = this->_right->evaluateString();

                for (unsigned i = 0; i < leftString.size(); i++) {
                    leftString[i] = tolower(leftString[i]);
                }
                for (unsigned i = 0; i < rightString.size(); i++) {
                    rightString[i] = tolower(rightString[i]);
                }

                int cmp = strcmp(leftString.c_str(), rightString.c_str());

                if (cmp == 0) {
                    result = true;
                }else{
                    result = false;
                }
                break;
            }
          default:
                warning("Runtime error: incompatible types of parameters for ",
                                "Equal operator");
        }
```

Figura 6.9

En casa caso observamos que se procesa de una forma diferente, si son números devolvemos simplemente el valor de la resta, si es 0 será verdadero, si no, será falso. Para los booleanos simplemente comprobamos si son iguales con el operador de igualdad de C++. Por último para las cadenas

comprobamos sin tener en cuenta las mayúsculas si todas las letras son iguales.

Para los demás operadores relacionales se hace un tratamiento parecido, solo cambiando el operador en cuestión. Cabe destacar que las cadenas se compara según el orden alfabético y sin tener en cuenta las mayúsculas.

Operadores lógicos: como todos los demás sigue la estructura de la clase operador además de añadir la función getType, tenemos dos operadores lógicos, “y (and)” y “o (or)”. Ambos tienen la misma estructura:

```
class AndNode : public LogicalOperatorNode
{
  public:

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

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

/*!
    \brief   Evaluate the AndNode
    \return  bool
    \sa      print()
*/
  bool evaluateBool();
};
```

Figura 6.10

La función evaluateBool es diferente en ambos, y lo que se realiza simplemente es utilizar los operadores de C++ and y or:

```
bool lp::AndNode::evaluateBool()
{
    bool result = false;

    if (this->getType() == BOOL)
    {
        bool leftBool, rightBool;

        leftBool = this->_left->evaluateBool();
        rightBool = this->_right->evaluateBool();

        result = leftBool and rightBool;
    }
    else
    {
        warning("Runtime error: incompatible types of parameters for ", "operator And");
    }

    return result;
}
```

Figura 6.11

```
bool lp::OrNode::evaluateBool()
{
    bool result = false;

    if (this->getType() == BOOL)
    {
        bool leftBool, rightBool;

        leftBool = this->_left->evaluateBool();
        rightBool = this->_right->evaluateBool();

        result = leftBool or rightBool;
    }
    else
    {
        warning("Runtime error: incompatible types of parameters for ", "operator Or");
    }

    return result;
}
```

Figura 6.12

Operadores de cadenas: nos encontramos con solo un operador específico para las cadenas, el cual es la concatenación. Aquí la clase:

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

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

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

Figura 6.13

El método evaluateString concatena las dos cadenas del nodo con el operador de suma de cadenas de C++.

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

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

  return result;
}
```

Figura 6.14

- ❏ UnaryOperatorNode es la clase padre de los dos tipos operadores unarios que tenemos, lógicos y numéricos:

```
class UnaryOperatorNode : public ExpNode
{
 protected:
  ExpNode *_exp;  //!< Child expression

 public:

/*!
    \brief Constructor of UnaryOperatorNode links the node to it child,
           and stores the character representation of the operator.
    \param expression: pointer to ExpNode
    \post  A new OperatorNode is created with the parameters
    \note  Inline function
*/
  UnaryOperatorNode(ExpNode *expression)
    {
        this->_exp = expression;
    }

    /*!
    \brief   Get the type of the child expression
    \return  int
    \sa      print
    */
    inline int getType()
    {
        return this->_exp->getType();
    }
};
```

Figura 6.15

Como vemos su variable es el nodo expresión sobre el que se aplicará el operador y la función getType que devuelve el tipo de la expresión.

Operadores unarios lógicos: tenemos un operador lógico unario, el operador de negación not.

```
class NotNode : public LogicalUnaryOperatorNode
{

 public:

/*!
    \brief Constructor of NotNode uses LogicalUnaryOperatorNode's constructor as member initializer
    \param expression: pointer to ExpNode
    \post  A new NotNode is created with the parameter
*/
  NotNode(ExpNode *expression): LogicalUnaryOperatorNode(expression)
    {
        // empty
    }

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

/*!
    \brief   Evaluate the NotNode
    \return  bool
    \sa      print()
*/
  bool evaluateBool();
};
```

Figura 6.16

Si miramos su función de evaluación:

```
bool lp::NotNode::evaluateBool()
{
    bool result = false;

    if (this->getType() == BOOL)
    {
        result = not this->_exp->evaluateBool();
    }
    else
    {
        warning("Runtime error: incompatible types of parameters for ", "operator Not");
    }

    return result;
}
```

Figura 6.17

Simplemente aplicamos el operador not de C++.

Operadores unarios numéricos: tenemos dos operadores unarios, menos y más, los dos tienen la estructura del operador unario lógico pero cambiando la función de evaluación por la evaluación numérica.

```
class UnaryMinusNode : public NumericUnaryOperatorNode
{

 public:

/*!
    \brief Constructor of UnaryMinusNode uses NumericUnaryOperatorNode's constructor as member initializer.
    \param expression: pointer to ExpNode
    \post  A new UnaryMinusNode is created with the parameter
    \note  Inline function: the NumericUnaryOperatorNode's constructor is used ad member initializer
*/
  UnaryMinusNode(ExpNode *expression): NumericUnaryOperatorNode(expression)
    {
        // empty
    }

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

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

Figura 6.18

La función de evaluación funciona de la siguiente forma:

```
double lp::UnaryMinusNode::evaluateNumber()
{
    double result = 0.0;

    // Ckeck the type of the expression
    if (this->getType() == NUMBER)
    {
        // Minus
        result = - this->_exp->evaluateNumber();
    }
    else
    {
        warning("Runtime error: the expressions are not numeric for ", "UnaryMinus");
    }

  return result;
}
```

Figura 6.19

La función para el más es igual solo que se le aplica el operador unario de suma de C++.

- ❏ VariableNode: clase que representa los nodos de variables.

```
class VariableNode : public ExpNode
{
    private:
      std::string _id; //!< Name of the VariableNode

    public:

    /*!
        \brief Constructor of VariableNode
        \param value: double
        \post  A new VariableNode is created with the name of the parameter
        \note  Inline function
    */
      VariableNode(std::string const & value)
        {
            this->_id = value;
        }

    /*!
        \brief   Type of the Variable
        \return  int
        \sa      print
    */
     int getType();

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

    /*!
        \brief   Evaluate the Variable as NUMBER
        \return  double
        \sa      print
    */
      double evaluateNumber();

    /*!
        \brief   Evaluate the Variable as BOOL
        \return  bool
        \sa      print
    */
      bool evaluateBool();

  /*!
        \brief   Evaluate the Variable as STRING
        \return  string
        \sa      print
    */
      std::string evaluateString();

};
```

Figura 6.20

Se guarda el identificador de la variable previamente instalado en la tabla de símbolos por el analizador léxico.
Por funciones tiene el constructor, getType para obtener el tipo de la variable, print para mostrar la información del nodo, y tres funciones de evaluación para poder obtener el valor de la variable tenga el tipo que tenga.

❏ NumberNode: nodo para representar los números.

```
class NumberNode : public ExpNode
{
 private:
   double _number; //!< \brief number of the NumberNode

 public:

/*!
    \brief Constructor of NumberNode
    \param value: double
    \post  A new NumberNode is created with the value of the parameter
    \note  Inline function
*/
  NumberNode(double value)
    {
        this->_number = value;
    }

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

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

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

Figura 6.21

Se guarda el valor del número representado por el nodo.
Tiene las funciones genéricas de todas las clases. La función de evaluación devuelve el valor numérico del nodo.

❏ ConstantNode: nodo para representar las constantes.

```
class ConstantNode : public ExpNode
{
    private:
      std::string _id; //!< Name of the ConstantNode

    public:

    /*!
        \brief Constructor of ConstantNode
        \param value: double
        \post  A new ConstantNode is created with the name of the parameter
    */
      ConstantNode(std::string value)
        {
            this->_id = value;
        }

    /*!
        \brief   Type of the Constant
        \return  int
        \sa      print
    */
     int getType();

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

    /*!
        \brief   Evaluate the Constant as NUMBER
        \return  double
        \sa      print
    */
      double evaluateNumber();

    /*!
        \brief   Evaluate the Constant as BOOL
        \return  bool
        \sa      print
    */
      bool evaluateBool();
};
```

Figura 6.22

Misma estructura que las demás clases. No incluye función de evaluación de cadenas puesto que no tenemos ninguna constante de este tipo.

- ❏ StringNode: clase para representar nodos cadena.

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

 public:

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

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

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

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

Figura 6.23

Idéntico al nodo numérico pero tratando el contenido como una cadena de C++.

- Statement: Clase padre de los clases que representan los nodos de las sentencias de control.

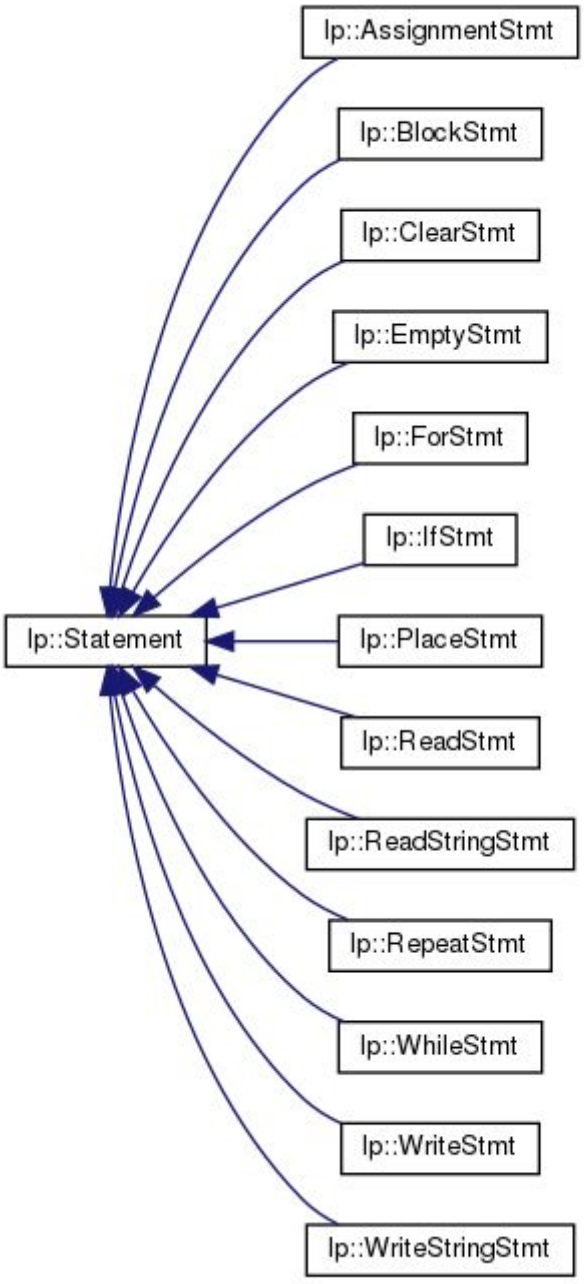


Figura 6.24

La estructura de la clase es la siguiente:

```
class Statement {
 public:

/*!
    \brief   Print the Statement
    \note    Virtual function: can be redefined in the heir classes
    \return  double
    \sa      print
*/

  virtual void print() {}

/*!
    \brief   Evaluate the Statement
    \warning Pure virtual function: must be redefined in the heir classes
    \return  double
    \sa      print
*/
  virtual void evaluate() = 0;
};
```

Figura 6.25

Define la funciones virtuales print y evaluate para que cada hijo las implemente como sea necesario.

Los hijos de esta clase son:

- ❏ AssignmentStmt: sentencia de asignación, asigna un valor a una variable.

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

  AssignmentStmt *_asgn;  //!< Allow multiple assigment -> a = b = 2

 public:

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

/*!
    \brief Constructor of AssignmentStmt
    \param id: string, variable of the AssignmentStmt
    \param asgn: pointer to AssignmentStmt
    \post  A new AssignmentStmt is created with the parameters
    \note  Allow multiple assigment -> a = b = 2
*/

  AssignmentStmt(std::string id, AssignmentStmt *asgn): _id(id), _asgn(asgn)
    {
        this->_exp = NULL;
    }


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

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

};
```

Figura 6.26

Como variables privadas tiene el identificador de la variable, la expresión que se asignará a la variable y un assigmentStmt para permitir asignaciones concatenadas.
Como métodos tiene dos constructores, uno para la asignación simple y otra para la compuesta. También dispone del reglamentario print y evaluate.

Si observamos el método evaluate:

```
void lp::AssignmentStmt::evaluate()
{

    lp::Variable *firstVar = (lp::Variable *) table.getSymbol(this->_id);

    // Check the expression
    if (this->_exp != NULL)
    {
        // Check the type of the expression of the asgn
        switch(this->_exp->getType())
        { ...
        }

    }

    ////////////////////////////////////////////
    // Allow multiple assigment -> a = b = c = 2;

    else // this->_asgn is not NULL
    {
        // IMPORTANT
        //  evaluate the assigment child
        this->_asgn->evaluate();


        /* Get the identifier of the previous asgn in the table of symbols as Variable */
        lp::Variable *secondVar = (lp::Variable *) table.getSymbol(this->_asgn->_id);

        // Get the type of the variable of the previous asgn
        switch(secondVar->getType())
        { ...
        }
    }
}
```

Figura 6.27

Primero se comprueba si es una asignación simple o compuesta comprobando hay expresión o no, ya que si no la hay es porque hay una segunda asignación.

```
switch(this->_exp->getType())
{
    case NUMBER:
    {
        double value;
        // evaluate the expression as NUMBER
        value = this->_exp->evaluateNumber();

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

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

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

    case BOOL:
    { ...
    }
    break;
    case STRING:
    { ...
    }
    break;

    default:
        warning("Runtime error: incompatible type of expression for ", "Assigment");
}
```

Figura 6.28

Si es simple se comprueba el tipo de la expresión. Si la variable ya es de ese tipo se le asigna el valor, si no, se borra de la tabla de símbolos y se instala una nueva con el mismo id pero del tipo de la expresión y con el nuevo valor. Si se le intenta asignar otro tipo, como por ejemplo el tipo indefinido, da error.

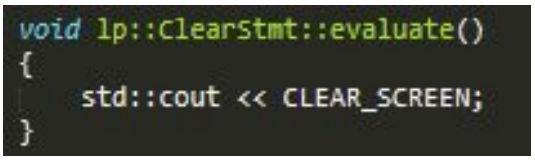
```
    else // this->_asgn is not NULL
    {
        // IMPORTANT
        //  evaluate the assigment child
        this->_asgn->evaluate();


        /* Get the identifier of the previous asgn in the table of symbols as Variable */
        lp::Variable *secondVar = (lp::Variable *) table.getSymbol(this->_asgn->_id);

        // Get the type of the variable of the previous asgn
        switch(secondVar->getType())
        {
            case NUMBER:
            {
                /* Get the identifier of the previous asgn in the table of symbols as NumericVariable */
                lp::NumericVariable *secondVar = (lp::NumericVariable *) table.getSymbol(this->_asgn->_id);
                // Check the type of the first variable
                if (firstVar->getType() == NUMBER)
                {
                /* Get the identifier of the first variable in the table of symbols as NumericVariable */
                lp::NumericVariable *firstVar = (lp::NumericVariable *) table.getSymbol(this->_id);
                    // Get the identifier o f the in the table of symbols as NumericVariable
//                  lp::NumericVariable *n = (lp::NumericVariable *) table.getSymbol(this->_id);

                    // Assignment the value of the second variable to the first variable
                    firstVar->setValue(secondVar->getValue());

                }
                // The type of variable is not NUMBER
                else
                {
                    // Delete the first variable from the table of symbols
                    table.eraseSymbol(this->_id);

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

            case BOOL:
            { ...
            }
            break;
            case STRING:
            { ...
            }
            break;
            default:
                warning("Runtime error: incompatible type of expression for ", "Assigment");
        }
    }
```

Figura 6.29

Si es una asignación compuesta se ejecuta la asignación de la derecha y después se le asigna el valor de la segunda variable a la primera.

- ❏ BlockStmt: esta sentencia no se usa en nuestro trabajo.

- ❏ ClearStmt: sentencia que limpia la pantalla.



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

Figura 6.30

El método evaluate usa el macro CLEAR_SCREEN para limpiar la pantalla

- ❏ EmptyStmt: clase que representa la sentencia vacía. No hace nada.
- ❏ ForStmt: clase que representa la sentencia for.

```
class ForStmt : public Statement
{
 private:
  std::string _identifier; //identifier
  ExpNode *_exp1; //1º expresion
  ExpNode *_exp2; //2º expresion
  ExpNode *_stepExp; //step expresion
  std:: list <Statement*> *_stmt; //!< Statement of the body of the repeat loop

  public:
/*!
    \brief Constructor of  ForStmt
    \param condition: ExpNode of the condition
    \param statement: Statement of the body of the loop
    \post  A new ForStmt is created with the parameters
*/
  ForStmt(std::string identifier, ExpNode *exp1, ExpNode *exp2, ExpNode *stepExp, std:: list <Statement*> *statement) : _identifier(identifier)
    {
    this->_exp1 = exp1;
    this->_exp2 = exp2;
    this->_stepExp = stepExp;
        this->_stmt = statement;
    }

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

Figura 6.31

Se guarda el identificador de la variable y las dos expresiones que guiarán la ejecución. También se guarda la expresión step, que indicará en cuanto se aumentará o decremantará la variable. Por último guardamos la lista de expresiones que se ejecutarán dentro del bucle.
Tiene dos constructores, uno por si se usa con step y otro para si se usa sin él.

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

        //check if _identifier is a number
        if (var->getType() != NUMBER)
        {
            table.eraseSymbol(this->_identifier);

            // Insert the variable in the table of symbols as NumericVariable
            // with the type NUMBER and the value
            lp::NumericVariable *v = new lp::NumericVariable(this->_identifier, VARIABLE, NUMBER, 0);
            table.installSymbol(v);
        }

        int step;
        //check if step is NULL
        if (this->_stepExp != NULL) {
            step = this->_stepExp->evaluateNumber();
        }else{
            //if step doesn't exists, put the default value of one
            step = 1;
        }

        if (step == 0) {
            warning("Infinte loop in for: ", "step equals zero");
        }else
        if ((this->_exp1->evaluateNumber() > this->_exp2->evaluateNumber()) and step > 0 ) {
            warning("Infinte loop in for: ", "'from' greater than 'to' and step is positive");
        }else
        if ((this->_exp1->evaluateNumber() < this->_exp2->evaluateNumber()) and step < 0 ) {
            warning("Infinte loop in for: ", "'to' greater than 'from' and step is negative");
        }else{
            //Numeric variable to modify the identifier
            lp::NumericVariable *v = (lp::NumericVariable *) table.getSymbol(this->_identifier);
            //for loop
            //from _exp1 to _exp2 with step "step"
            for (int a = this->_exp1->evaluateNumber(); a <= this->_exp2->evaluateNumber(); a = a + step) {
                //modify the identifier in every step
                v->setValue(a);
                std::list<Statement*>::iterator i=this->_stmt->begin();
                //execute the list of statements
                while(i!=_stmt->end()) {
                    (*i)->evaluate();
                    i++;
                }
            }
        }
    }

}
```

Figura 6.32

Primero comprobamos si las expresiones son numéricas. Si lo son pasamos a comprobar si la variable es numérica y en caso de no serla se le pasa el valor a numérico con el valor de inicio. Comprobamos también si el bucle es infinito. Si todas estas comprobaciones son pasadas se ejecuta como un for de C++.

❏ IfStmt: sentencia condicional if.

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


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

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

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

Figura 6.33

Tiene cuatro constructores para los casos en los que tiene else y no y para los casos en los que la condición es una constante.

```
void lp::IfStmt::evaluate()
{

    //check if the condition is boolean
    if (this->_cond->getType() != BOOL) {
        warning("Runtime error: incompatible types for", "if condition");
    }else{
        // If the condition is true,
        if (this->_cond->evaluateBool() == true ){
            // the consequent is run
            std::list<Statement*>::iterator i=this->_stmt1->begin();

            while(i!=_stmt1->end()) {
                (*i)->evaluate();
                i++;
            }
        }
        // Otherwise, the alternative is run if exists
        else if (this->_stmt2 != NULL){
            std::list<Statement*>::iterator i=this->_stmt2->begin();
            while(i!=_stmt2->end()) {
                (*i)->evaluate();
                i++;
            }
        }
    }
}
```

Figura 6.34

Se ejecuta como if normal después de comprobar si la condición es booleana.

- ❏ PlaceStmt: sentencia que coloca el cursor en las coordenadas especificadas.

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

Figura 6.35

Se usa la función PLACE con los valores obtenidos.

- ❏ ReadStmt: clase para leer el valor de una variable numérica por teclado.

```
void lp::ReadStmt::evaluate()
{
    std::string stringAux;
    double value;
    fflush(stdin);
    std::cin >> stringAux;
    std::cin.ignore();

    value = atof(stringAux.c_str());

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

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

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

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

        table.installSymbol(n);
    }
}
```

Figura 6.36

Se lee por pantalla una variable de C++ string y se pasa a numérica, depués comprobamos si el nodo variable es de tipo numérico para introducirle el valor, si no lo es se instala en la tabla de símbolos como numérica.

- ReadStringStmt: sentencia que lee una cadena desde teclado.

```
void lp::ReadStringStmt::evaluate()
{
    fflush(stdin);
    char stringAux[100];
    std::cin.getline(stringAux, 100);
    fflush(stdin);
    /* Get the identifier in the table of symbols as Variable */
    lp::Variable *var = (lp::Variable *) table.getSymbol(this->_id);

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

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

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

        table.installSymbol(n);
    }
}
```

Figura 6.37

Se sigue un método parecido al de ReadStmt pero leyendo una cadena e introduciendo la variable como cadena.

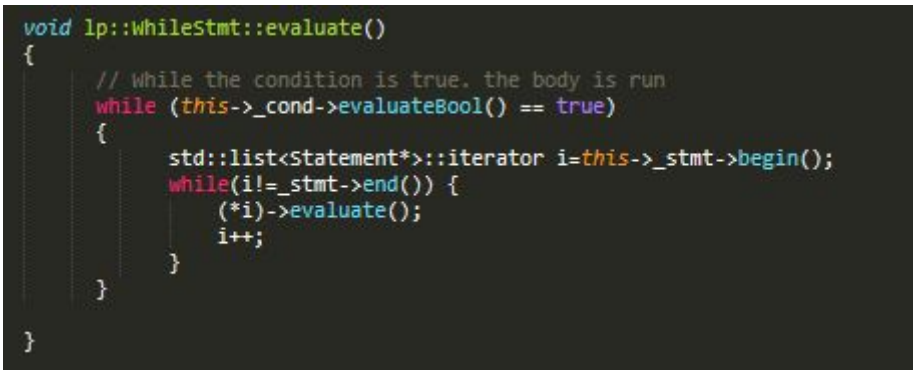
- ❏ RepeatStmt: bucle repetir hasta que.

```
void lp::RepeatStmt::evaluate()
{
      // While the condition is true. the body is run
      do
      {
            std::list<Statement*>::iterator i=this->_stmt->begin();
            while(i!=_stmt->end()) {
                (*i)->evaluate();
                i++;
            }
      }while(this->_cond->evaluateBool() == false);
}
```

Figura 6.38

Se implementa como un do while de C++ pero negando la condición.

- ❏ WhileStmt: bucle mientras.

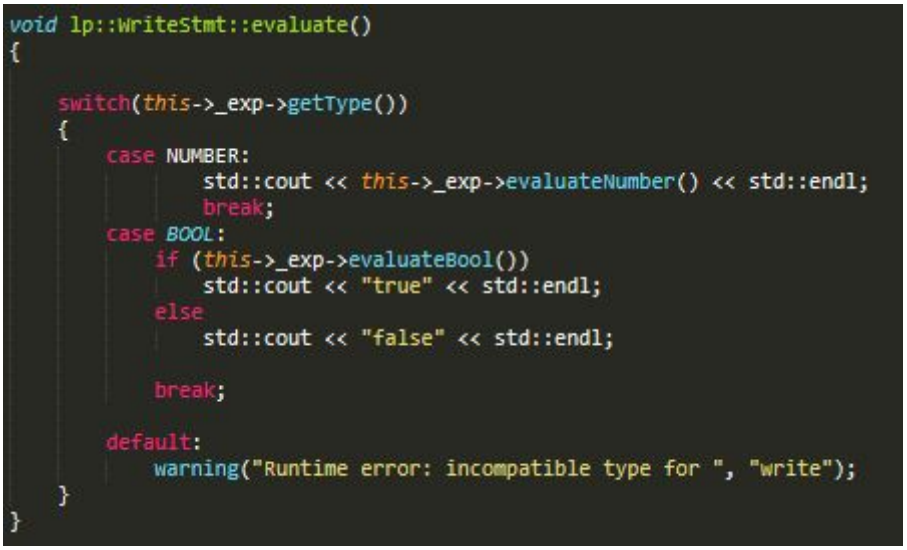


```
void lp::WhileStmt::evaluate()
{
      // While the condition is true. the body is run
      while (this->_cond->evaluateBool() == true)
      {
            std::list<Statement*>::iterator i=this->_stmt->begin();
            while(i!=_stmt->end()) {
                (*i)->evaluate();
                i++;
            }
      }

}
```

Figura 6.39

La implementación es un while de C++.

- WriteStmt: sentencia para escribir una variable numérica o booleana por pantalla.



```
void lp::WriteStmt::evaluate()
{

    switch(this->_exp->getType())
    {
        case NUMBER:
                std::cout << this->_exp->evaluateNumber() << std::endl;
                break;
        case BOOL:
            if (this->_exp->evaluateBool())
                std::cout << "true" << std::endl;
            else
                std::cout << "false" << std::endl;

            break;

        default:
            warning("Runtime error: incompatible type for ", "write");
    }
}
```

Figura 6.40

Se comprueba el tipo de la variable, si es numérica se imprime el valor y si es booleana se imprime su valor lógico en lenguaje humano.

- WriteStringStmt: sentencia para escribir una cadena por pantalla.

```
void lp::WriteStringStmt::evaluate()
{


    if (this->_exp->getType() == STRING) {
        std::string aux = this->_exp->evaluateString();
        for (unsigned i = 0; i < aux.size(); i++) {
            if (aux[i] != '\\') {
                std::cout << aux[i];
            }else{
                i++;
                switch (aux[i]) {
                    case 't':
                        std::cout << "\t";
                        break;
                    case 'n':
                        std::cout << "\n";
                        break;
                    case '\'':
                        std::cout << "'";
                        break;
                    default:
                        std::cout << "\\" <<aux[i];
                }
            }
        }
    }else{
        warning("Runtime error: incompatible type for ", "writeString");
    }

}
```

Figura 6.41

Se recorre la cadena y se va mostrando cada carácter, si es una barra “\” se comprueba si el siguiente carácter es una “t” para insertar una tabulación, una “n” para insertar un salto de línea o una “ ‘ “ para introducir una comilla simple, si no es ninguno de estos caracteres se imprime la barra y el carácter.

7.Funciones auxiliares

En principio nosotros no usamos funciones auxiliares ni clases auxiliares, esto se debe a que principalmente hemos realizado cambios sobre el propio código proporcionado por el profesor y hemos añadido aquello que nos hacía falta como es el caso de los String o incluso, por ejemplo, la función *clear*. Simplemente la hemos añadido junto al código proporcionado, no hemos visto necesario añadirlo en un fichero aparte ya que la mayoría de funciones poseían un tamaño bastante reducido.

Las funciones realizadas por nosotros se encuentran en el apartado 6, dentro de sus respectivas clases que es el Código de AST.

8.Modo de obtención del intérprete

En este apartado se van a explicar los directorios y los archivos que constituyen a nuestro intérprete, primeramente vamos a ver el directorio principal del intérprete:

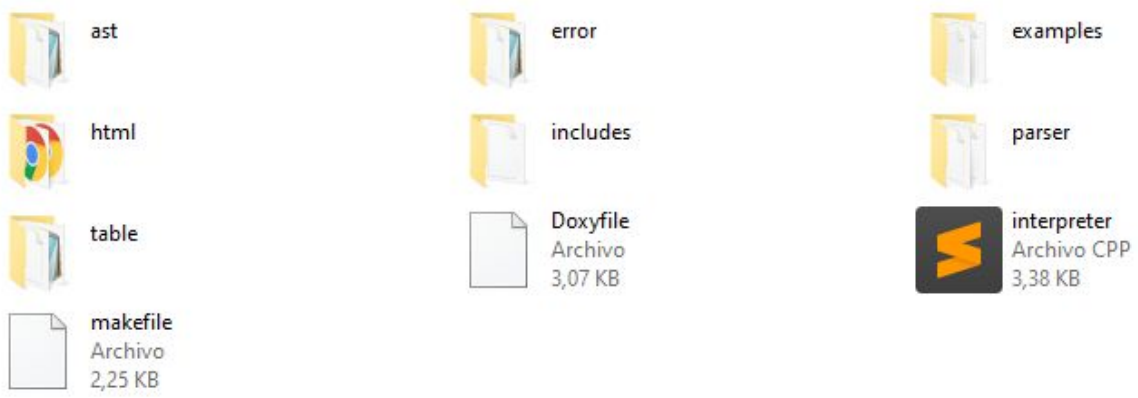


Figura 8.1

Como podemos observar encontramos los ficheros:

- **Doxyfile**. Este fichero nos permite crear un Doxygen del proyecto donde se explican todas las clases usadas. Principalmente es la documentación del Intérprete.
- **Makefile**. Es un fichero que nos permite hacer make al proyecto, es decir, permite la compilación de una forma más sencilla de todo el proyecto.
- **interpreter.cpp.** Es el fichero principal del programa, es decir, es el main del programa.

Respecto a los directorios nos encontramos los siguientes:

- **Ast.** En este directorio nos vamos a encontrar todas las clases especificadas y comentadas en el apartado 6. Por tanto dentro de este directorio hay:

Figura 8.2

-El fichero ast.cpp contiene el desarrollo de los métodos de las clases.
-El fichero ast.hpp contiene las definiciones de las clases.
-makefile es un fichero para compilar estos dos ficheros.

- **Error.** En este directorio nos encontramos con los ficheros de errores, estos ficheros contienen las funciones de error, esta funciones son usadas en otras clases y permiten ser usadas para dar mensajes de error personalizados. Dentro de este directorio encontramos:

Figura 8.3

-El fichero error.cpp que contiene el desarrollo de los métodos de la clase.
-El fichero ast.hpp que contiene la definición de la clase.
-makefile que es un fichero para compilar estos dos ficheros.

- **Examples**. En este directorio encontramos ejemplos que podemos pasar al intérprete y que este es capaz de usar. Estos tienen sentido cuando no se usa el modo interactivo del intérprete. Podemos encontrar los siguientes ficheros:

Figura 8.4

-El fichero ejemplo-bucle-for.p contiene un ejemplo de uso para el bucle for.
-El fichero ejemplo-bucle-repeat.p contiene un ejemplo de uso para el bucle repeat.
-El fichero ejemplo-bucle-while-clear-y-place.p contiene un ejemplo de uso para el bucle while y los comandos clear y place.
-El fichero ejemplo-con-errores.p es un fichero donde mostramos como funciona nuestro intérprete ante ciertos fallos que se le pueden llegar a pasar como por ejemplo bucles infinitos o identificadores mal escritos.
-El fichero ejemplo-if.p contiene un ejemplo de uso para if.
-El fichero ejemplo-operadores-logicos-y-cadenas.p es un fichero que contiene un ejemplo de uso de las cadenas y de los operadores lógicos.

-El fichero ejemplo-operadores-numericos.p contiene un ejemplo con los operadores numéricos y cómo se usan estos.
-El fichero ejemplo-operadores-relacionales.p contiene un ejemplo de uso de los operadores relacionales.
-El fichero ejemplo-read-write-readstring-writestring.p contiene un ejemplo de cómo se usan todas estas sentencias.

Todos estos ficheros se ven más detallados en el apartado 10, Ejemplos.

- **html**. En este directorio encontramos toda la documentación de doxygen, para acceder a la misma solo hay que hacer click en el fichero index que se encuentra dentro de este directorio.

- **includes**. En este directorio encontramos los includes que usa nuestro intérprete y que son externos a él. El fichero que encontramos es el siguiente:

Figura 8.5

-El fichero macros.hpp tiene macros que son utilizados por ciertas funciones de nuestro interprete, estos por ejemplo son utilizados en los errores personalizados dentro del fichero error.cpp.

- **parser**. En este directorio encontramos el fichero de flex y el fichero de bison, es decir, en este directorio se encuentra el análisis léxico y el análisis sintáctico que contiene nuestro intérprete. Los ficheros que encontramos son los siguientes:

Figura 8.6

-El fichero interpreter.l es el léxico de nuestro intérprete, ahí encontramos todo lo relacionado al léxico ya que es nuestro fichero de flex. Su interior está explicado en el apartado 4, Análisis léxico.

-El fichero interpreter.y es el sintáctico de nuestro intérprete, ahí encontramos todo lo relacionado al sintáctico ya que es nuestro fichero de bison. Su interior está explicado en el apartado 5, Análisis sintáctico.
-makefile es un fichero para compilar estos dos ficheros.

- **table**. En este directorio encontramos todas las clases relacionadas con el siguiente esquema:

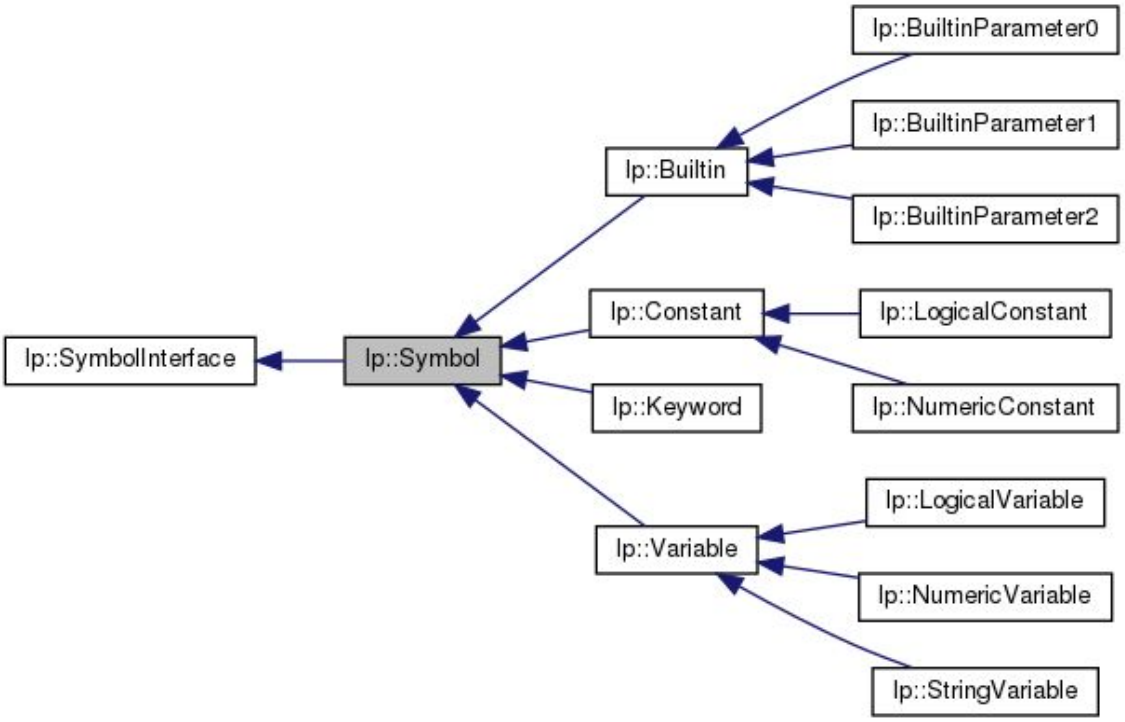


Figura 8.7

Como podemos observar, dentro de este directorio nos vamos a encontrar estas clases en sus respectivos ficheros junto a las clases relacionadas con table explicadas en el apartado 3, Tabla de símbolos.

Dentro de la carpeta nos encontramos:

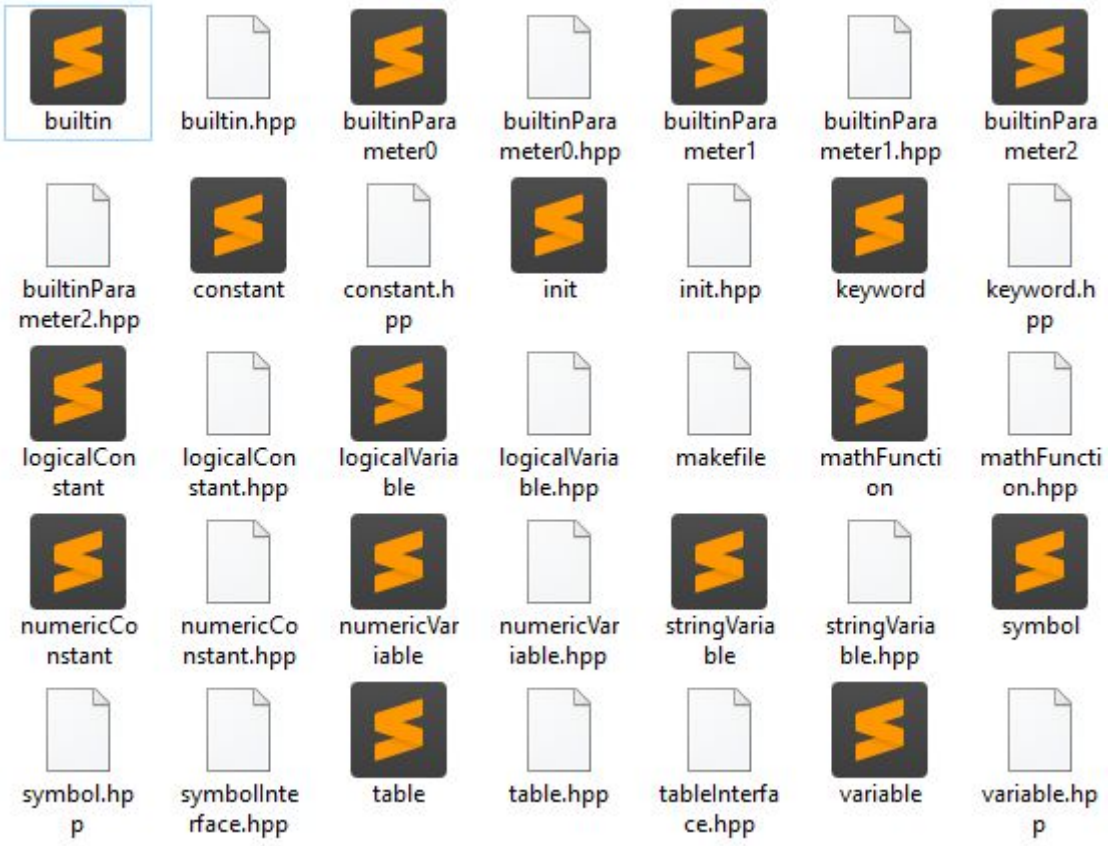


Figura 8.8

-Ficheros builtin.cpp y builtin.hpp que contiene a la clase builtin.
-Ficheros builtinParameter0.cpp y builtinParameter0.hpp que contiene a la clase builtinParameter0.
-Ficheros builtinParameter1.cpp y builtinParameter1.hpp que contiene a la clase builtinParameter1.
-Ficheros builtinParameter2.cpp y builtinParameter2.hpp que contiene a la clase builtinParameter2.
-Ficheros constant.cpp y constant.hpp que contiene a la clase constant.
-Ficheros init.cpp y init.hpp que contiene a la clase init. En esta clase encontramos los valores constantes inicializados, los que vienen dados por los tipos de la clase numeric, logical, keyword y función dados por las clases builtin. Cabe destacar que en init.hpp se encuentran las palabras clave creadas.
-Ficheros keyword.cpp y keyword.hpp que contiene la clase keyword.
-Ficheros logicalConstant.cpp y logicalConstant.hpp que contiene la clase logicalConstant.
-Ficheros logicalVariable.cpp y logicalVariable.hpp que contiene la clase logicalVariable.
-makefile es un fichero para compilar todos estos ficheros.
-Ficheros mathFuncion.cpp y mathFuncion.hpp que contiene la clase mathFuncion.
-Ficheros numericConstant.cpp y numericConstant.hpp que contiene la clase numericConstant.

-Ficheros numericVariable.cpp y numericVariable.hpp que contiene la clase numericVariable.
-Ficheros stringVariable.cpp y stringVariable.hpp que contiene la clase stringVariable.
-Ficheros symbol.cpp y symbol.hpp que contiene la clase symbol.
-Ficheros symbolInterface.hpp que contiene la clase symbolInterface.
-Ficheros table.cpp y table.hpp que contiene la clase table.
-Ficheros tableInterface.hpp que contiene la clase tableInterface.
-Ficheros variable.cpp y variable.hpp que contiene la clase variable.

9.Modo de ejecución del intérprete

Tras crear el intérprete este puede funcionar de dos maneras, de modo interactivo o a partir de un fichero.
Respecto al **modo interactivo**, el usuario se comunica directamente con el intérprete, pasándole por consola aquello que quiere que el intérprete reconozca.
Respecto al **modo no interactivo**, el usuario le pasa por línea de comandos un fichero, si este es válido, lo leerá y analizará las partes del mismo.
Para ver en qué modo funcionara, se realiza en el main (interpreter.cpp) una comprobación como podemos observar:

```
if (argc == 2)
{
    std::string filename(argv[1]);
    if (filename.size()<3) {
      std::cerr << "File's name not valid" << '\n';
      exit(-1);
    }
    if ((filename[filename.size()-1] != 'p') or (filename[filename.size()-2] != '.')) {
      std::cerr << "The file must have '.p' extension" << '\n';
      exit(-1);
    }
    yyin = fopen(filename.c_str(),"r");
    if (yyin == NULL) {
      std::cerr << "Input file does not exist" << '\n';
      exit(-1);
    }


  interactiveMode = false;
}
else
{
 interactiveMode = true;
}
```

Figura 9.1

Si no se le introduce nada por línea de comandos, entonces el modo es interactivo, en el caso contrario, si se le manda un fichero válido, el modo interactivo será false.

Cabe destacar que si el modo no es interactivo, en el interpreter.l hay una regla donde permite que tras coger todos los elementos el léxico pare, esto se consigue gracias a la siguiente regla:

```
<<EOF>> { /* The interpreter finishes when finds the end of file character */
        /*  PLACE(24,10);
          std::cout <<  BICYAN;
          std::cout <<  ">>>>>>> End of file <<<<<<<";
          std::cout << std::endl;
          std::cout << RESET;
          PLACE(25,1);
        */
          return 0;
    }
```

Figura 9.2

Por último, cabe enseñar un caso no trivial que ocurre a la hora del funcionamiento cuando es el modo interactivo, es el siguiente:

```
    /*  NEW in example 17 */
if: /* Simple conditional statement */
  IF controlSymbol LPAREN CONSTANT RPAREN THEN stmtlist ENDIF
  {
    // Create a new if statement node
    lp::ConstantNode *aux = new lp::ConstantNode($4);
    $$ = new lp::IfStmt(aux, $7);
    // To control the interactive mode
    control--;
    warning("warning: useless if:", "condition is constant");
  }
```

Figura 9.3

Esto es un ejemplo, en otros casos también ocurre los mismo, como por ejemplo while o for. Esto se debe a que como dentro de ellos pueden tener listas de sentencias, se debe permitir en modo interactivo que mientras no termine el while o el for, puede poner tantas como quisiera, de tal modo que posteriormente se puedan evaluar cada una de ellas.

10.Ejemplos:

Hemos codificado 9 ejemplos en total, aparte de los 2 proporcionados por el profesor, mostrando el uso de varias sentencias de control y expresiones. Procedemos a explicar brevemente cada uno de ellos:

1. **ejemplo-read-write-readstring-writestring.p:** en este ejemplo se muestra el uso básico de las sentencias de lectura y escritura del pseudocódigo.
2. **ejemplo-operadores-numericos.p:** en este fichero se encuentran varios pequeños ejemplos del uso de nuestros operadores numéricos.
3. **ejemplo-operadores-relacionales.p:** muestra del uso de los operadores relacionales, tanto con variables y constantes numéricas como con variables y constantes alfanuméricas.
4. **ejemplo-operadores-lógicos-y-cadenas.p:** en este caso mostramos el uso de los operadores lógicos y del único operador de cadenas.
5. **ejemplo-if.p:** en este fichero se muestran ejemplos de uso del if, además de una pequeña demostración de que el intérprete no distingue entre mayúsculas y minúsculas.
6. **ejemplo-bucle-for.p:** ejemplo del bucle for realizando un pequeño programa sobre la sucesión de fibonacci.
7. **ejemplo-bucle-while.p:** en este fichero se ha codificado el cálculo del factorial de un número para demostrar el uso del bucle while.
8. **ejemplo-bucle-repeat.p:** hemos codificado un pequeño juego para demostrar el uso del bucle repeat.
9. **ejemplo-con-errores.p:** una lista de sentencias y expresiones que contienen errores para mostrar el tratamiento de estos.
10. **menu.p:** ejemplo de un menú con un par de opciones.
11. **conversion.p:** ejemplo de la conversión de tipos en variables.

11.Diario:

En este apartado, hemos realizado un diario donde hemos recogido lo que íbamos realizando cada día que avanzabamos en el trabajo.

11/05/2020
-Creación de palabras reservadas (las mayúsculas y minúsculas no se diferencian).
-Creación de identificadores (no se controlan errores).
-Creación de números
-Creación de operadores aritméticos.
-Modificación de operadores lógicos

18/05/2020
-Operador de asignación
-Operadores aritméticos
-Operadores relacionales
-Creación de operadores aritméticos (arreglar fallos de código e implementar quotient).
-HACER: cadenas

21/05/2020
-Sentencia de control IF
-Sentencia de control WHILE
-Sentencia de control REPEAT
-Empezamos sentencia de control for

25/05/2020
-Terminamos la sentencia de control for
-Comandos especiales Clear
-Comandos especiales Place

26/05/2020
-Comentarios Terminados
-Comenzando la creación de las cadenas y operador de concatenación (hecho)
-Terminamos WriteString

28/05/2020

-Terminamos ReadString
-Terminamos Operadores relacionales para cadenas
-Cadenas terminadas
-Control de errores léxicos.
-Hacer main
-Arreglar errores varios
-Crear reglas para controlar errores

03/06/2020

-Terminamos de arreglar últimos problemas con las cadenas

12.Conclusiones:

Reflexión sobre el trabajo realizado:

El trabajo nos ha ayudado a comprender mejor la teoría sobre analizadores léxico-sintácticos. Hemos comprendido cómo se aplican las reglas de producción en un entorno real y cómo se codifican estas. También hemos podido entender cómo realizar una recuperación de errores y el trabajo de reflexión e imaginación que esto conlleva al tener que pensar e imaginar los casos en los que el intérprete tiene que reconocer un error en el lenguaje.
Aunque nuestro trabajo es bastante simple ya que más de la mitad del código ya estaba codificado y la estructura general ya nos la dieron, nos deja una idea de lo complicado y trabajoso que resulta codificar un intérprete.
En resumen, nos ha ayudado a comprender cómo funcionan los intérpretes por dentro, a comprender mejor la teoría estudiada en clase y ya no vemos de la misma forma los mensajes que nos dan los compiladores de otros lenguajes

Puntos fuertes y puntos débiles del intérprete:

-Los principales puntos fuertes son:

- Nuestro intérprete **no tiene el problema del else danzante** gracias a que se usa endif, esto se puede aplicar al resto de bucles ya que nos ayuda a evitar problemas similares del mismo estilo.
- Nuestro **bucle for es bastante robusto a los errores**, comprobando los casos en los que el bucle llegaría a ser infinito.
- Es **ampliable**, siempre que se quiera se pueden añadir nuevas reglas al léxico o nuevas sentencias al semántico para mejorarlo y que el intérprete abarque más funcionalidades y sea mucho más robusto.
- **No hay cascada de errores**, esto permite que sea mucho más sencillo entender dónde se encuentra el fallo y que así el programador decida cómo arreglarlo.

-Los principales puntos débiles son:

- **No hemos encontrado ninguno destacable** pero cabe destacar que nuestro intérprete **es mejorable** ya que se puede expandir con **operadores más complejos** como la suma/resta con asignación (+:= o -:=) y derivados, también se podrían añadir **nuevas sentencias de control** como el switch, por último, aunque nuestro programa detecta bastante errores, se podría mejorar el hecho de informar al usuario sobre ciertos errores concretos.

13.Bibliografía y referencias web

- Kernighan, B. W. y Pike, R. (1984). The Unix programming enviroment. New Jersey: Prentice Hall. ISBN: 0-13- 937699-2

- Levine, J. R.; Mason, T. y Brown, D.(1992). Lex & Yacc. Sebastopol (California): O'Reilly & Associates, Inc. ISBN: 1-56592-000-7

- La página web cplusplus donde hay documentació sobre c++ https://www.cplusplus.com/

- La página web stackOverFlow, suele haber soluciones a la mayoría de dudas https://es.stackoverflow.com/

- Enlace a wikipedia para ver la Sucesión de Fibonacci https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci

14.Anexos

- Manual sobre **Flex** que hemos visto interesante http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.pdf
- Manual oficial de **Bison**, de la propia página web de GNU https://www.gnu.org/software/bison/manual/