
Córdoba, 28 de junio 2021

Intérprete de pseudocódigo en español: IPE

Jose María Cabrera Rivas (i82carij)

Luis Ballesteros Arévalo (i82baarl)

Grado en Ingeniería Informática

Especialidad: Computación

Segundo cuatrimestre

Escuela Politécnica Superior

Procesadores de Lenguajes

Tercer curso

Curso: 2020/2021

Universidad de Córdoba



UNIVERSIDAD DE CÓRDOBA



ESCUELA POLITÉCNICA
SUPERIOR DE CÓRDOBA
Universidad de Córdoba

Índice

| | |
|--|----|
| Índice | 1 |
| 1 Introducción | 3 |
| 2 Lenguaje de pseudocódigo | 3 |
| 2.1 Componentes léxicos o tokens | 3 |
| Palabras reservadas: | 3 |
| Identificadores: | 4 |
| Número | 4 |
| Cadena | 5 |
| Operador de asignación | 5 |
| Operadores aritméticos | 5 |
| Operador alfanumérico | 7 |
| Operadores relacionales de números y cadenas | 8 |
| Operadores lógicos | 8 |
| Comentarios | 8 |
| Punto y coma | 9 |
| 2.2 Sentencias | 9 |
| Asignación | 9 |
| Lectura | 9 |
| Escritura | 10 |
| Sentencias de control | 10 |
| Comandos especiales | 11 |
| Ampliaciones | 11 |
| 3 Tabla de símbolos | 12 |
| 4 Análisis léxico | 23 |
| 5 Análisis sintáctico | 27 |
| IF | 31 |
| WHILE | 32 |
| REPETIR | 33 |
| PARA | 34 |
| SWITCH | 36 |
| 6 Código de AST | 37 |
| 7 Modo de obtención del intérprete | 58 |
| AST | 58 |
| Error | 58 |

| | |
|---|-----------|
| Includes | 58 |
| Parser | 58 |
| Table | 58 |
| 8 Modo de ejecución del intérprete | 59 |
| Interactiva | 59 |
| A partir de un fichero | 59 |
| 9 Ejemplos | 60 |
| Conversion.e | 60 |
| Menu.e | 61 |
| Ejemplo 1 | 65 |
| Ejemplo 2 | 65 |
| Ejemplo 3 | 67 |
| 10 Conclusiones | 68 |
| 11 Bibliografía | 68 |

1 Introducción

En este trabajo, se nos pide la elaboración de un intérprete de pseudocódigo en español utilizando bison y flex.

El intérprete podrá analizar léxicamente, sintácticamente y semánticamente cualquier documento que cumpla los requisitos del [lenguaje de pseudocódigo](#).

Además de recibir y analizar documentos, podremos usar el modo interactivo desde el cual podremos introducir sentencias por la terminal y se podrán analizar de igual modo que un fichero.

Este documento se dividirá en varios apartados en los que iremos explicando cada apartado de la práctica detalladamente. Además aportaremos ejemplos en los que se mostrará de forma ilustrativa el funcionamiento de nuestro intérprete.

2 Lenguaje de pseudocódigo

Las características del pseudocódigo son las siguientes:

2.1 Componentes léxicos o tokens

Palabras reservadas:

Las palabras reservadas se guardan en el fichero init.hpp, Nuestro fichero tiene la siguiente estructura:

```
static struct {
    std::string name ;
    int token;
    } keyword[] = {
        {"escribir", PRINT},
        {"escribir_cadena", PRINT_CAD},
        {"leer", READ},
        {"leer_cadena", READ_CAD},

        {"si", IF}, // NEW in example 17
        {"si_no", ELSE}, // NEW in example 17
        {"fin_si", END_IF},
        {"entonces", THEN},

        {"mientras", WHILE}, // NEW in example 17
        {"hacer", DO},
        {"fin_mientras", END_WHILE},

        {"repetir", REPETIR},
        {"hasta", UNTIL},

        {"para", FOR},
        {"desde", FROM},
```

```

        {"paso", PASS},
        {"fin_para", END_FOR},

        {"casos", CASOS},
        {"valor", VALOR},
        {"fin_casos", FIN_CASOS},
        {"defecto", DEFECTO},

        {"#mod", MODULO},
        {"#div", DIVENTERA},
        {"#o", OR},
        {"#y", AND},
        {"#no", NOT},

        {"", 0}
    };

```

Identificadores:

Características:

- Estarán compuestos por una serie de letras, dígitos y el subrayado.
- Deben comenzar por una letra
- No podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.
- No se distingue entre mayúscula y minúscula.

Un ejemplo de un identificador sería:

```

dato_1 := 4;
assignment_node: =
    dato_1
    NumberNode: 4

```

El identificador es "dato_1" que como se puede observar, cumple todos los requisitos propuestos. Para los identificadores con errores, tenemos un control de errores que se explicará más adelante.

Número

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

Algunos ejemplos de su implementación:

```

numero := 3;
assignment_node: =
    numero
    NumberNode: 3

numero2 := 3.5;
assignment_node: =
    numero2
    NumberNode: 3.5

numero3 := 3e-5;
assignment_node: =
    numero3
    NumberNode: 3e-05

```

Como se puede ver, podemos utilizar tanto números decimales como notación científica para nuestro intérprete.

Cadena

- Estará compuesta por una serie de caracteres delimitados por comillas simples.
- Deberá permitir la inclusión de la comilla simple utilizando la barra (\)

```

cadena := 'esto es una cadena';
'esto es una cadena'assignment_node: =
    cadena
    CadNode: esto es una cadena (Type: 284)

```

Operador de asignación

asignación: :=

Operadores aritméticos

- suma: +
 - ◆ Unario: + 2
 - ◆ Binario: 2 + 3

```

escribir(3+2);
PrintStmt: print
    PlusNode: +
    NumberNode: 3
    NumberNode: 2

Print: 5

```

```

dato := +2;
assignment_node: =
    dato
    UnaryPlusNode: +
    NumberNode: 2

```

→ resta: -

- ◆ Unario: - 2
- ◆ Binario: 2 - 3

```
escribir(3-5);  
PrintStmt: print  
    MinusNode: -  
    NumberNode: 3  
    NumberNode: 5  
  
Print: -2
```

```
dato2:= -3;  
assignment_node: =  
    dato2  
    UnaryMinusNode: -  
    NumberNode: 3
```

→ producto: *

```
escribir(5*7);  
PrintStmt: print  
    MultiplicationNode: *  
    NumberNode: 5  
    NumberNode: 7  
  
Print: 35
```

→ división: /

```
escribir(10/3);  
PrintStmt: print  
    DivisionNode: /  
    NumberNode: 10  
    NumberNode: 3  
  
Print: 3.333333
```

→ división entera: #div

```
escribir(10 #div 3);  
PrintStmt: print  
    DivisionEnteraNode: /  
    NumberNode: 10  
    NumberNode: 3  
  
Print: 3
```

→ módulo: #mod

```
escribir(10 #mod 3);
PrintStmt: print
    ModuloNode: %
    NumberNode: 10
    NumberNode: 3

Print: 1
```

→ potencia: **

```
escribir(5 ** 4);
PrintStmt: print
    PotenciaNode: **
    NumberNode: 5
    NumberNode: 4

Print: 625
```

Operador alfanumérico

concatenación: ||

```
cadena1:= 'buenos ';
'buenos 'assignment_node: =
    cadena1
    CadNode: buenos (Type: 284)

cadena2:= 'dias';
'dias'assignment_node: =
    cadena2
    CadNode: dias (Type: 284)

escribir_cadena(cadena1||cadena2);
PrintStmt: print
    ConcatNode: +
    VariableNode: cadena1 (Type: 284)
    VariableNode: cadena2 (Type: 284)

Print: buenos dias
```

Operadores relacionales de números y cadenas

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

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

Operadores lógicos

- disyunción lógica: #o
- conjunción lógica: #y
- negación lógica: #no

Comentarios

- De varias líneas: delimitados por el símbolos << y >>

```
<< esto es un  
comentario  
de varias líneas  
>>  
Encontrado comentario de varias líneas
```

- De una línea
 - ◆ Todo lo que siga al carácter @ hasta el final de la línea.

```
@ esto es un comentario de una línea  
Encontrado comentario de una línea
```

Punto y coma

Se utilizará para indicar el fin de una sentencia

2.2 Sentencias

Asignación

- **identificador := expresión numérica**
 - ◆ Declara a identificador como una variable numérica y le asigna el valor de la expresión numérica.
 - ◆ Las expresiones numéricas se formarán con números, variables numéricas y operadores numéricos.
- **identificador := expresión alfanumérica**
 - ◆ Declara a identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.

-
- ◆ Las expresiones alfanuméricas se formarán con cadenas, variables alfanuméricas y el operador alfanumérico de concatenación (||).

Lectura

→ Leer (identificador)

- ◆ Declara a identificador como variable numérica y le asigna el número leído.

```
leer(id);  
ReadStmt: read  
    id  
Inserta un valor numerico --> 5
```

→ Leer_cadena (identificador)

- ◆ Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).

```
leer_cadena(cadena);  
ReadStringStmt: read  
    cadena  
Inserta una cadena --> buenos días
```

Escritura

→ Escribir (expresión numérica)

- ◆ El valor de la expresión numérica es escrito en la pantalla.

```
escribir(id);  
PrintStmt: print  
    VariableNode: id (Type: 283)  
Print: 5
```

```
escribir(2+3);  
PrintStmt: print  
    PlusNode: +  
    NumberNode: 2  
    NumberNode: 3  
Print: 5
```

→ **Escribir_cadena (expresión alfanumérica)**

- ◆ La cadena (sin comillas exteriores) es escrita en la pantalla.
- ◆ Se debe permitir la interpretación de comandos de saltos de línea (\n) y tabuladores (\t) que puedan aparecer en la expresión alfanumérica.

```
escribir_cadena(cadena);
PrintStmt: print
    VariableNode: cadena (Type: 284)
Print: buenos días
```

```
escribir_cadena('buenos días');
'buenos días'PrintStmt: print
    CadNode: buenos días (Type: 284)
Print: buenos días
```

Sentencias de control

→ Sentencia condicional simple

si condición
 entonces lista de sentencias
fin_si

→ Sentencia condicional compuesta

si condición
 entonces lista de sentencias
 si_no lista de sentencias
fin_si

→ Bucle mientras

mientras condición **hacer**
 lista de sentencias
fin_mientras

→ Bucle repetir

repetir
 lista de sentencias
hasta condición

→ Bucle2 “para”

para identificador

desde expresión numérica 1
hasta expresión numérica 2
[paso expresión numérica 3]
hacer lista de sentencias

fin_para

El **paso** es opcional; en su defecto, tomará el valor 1

→ Sentencia “casos”

casos (expresión)

valor v1: ...

valor v2:

defecto: ...

fin_casos

Comandos especiales




































→ **#borrar** : borra la pantalla

→ **#lugar**(expresión numérica1, expresión numérica2): Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

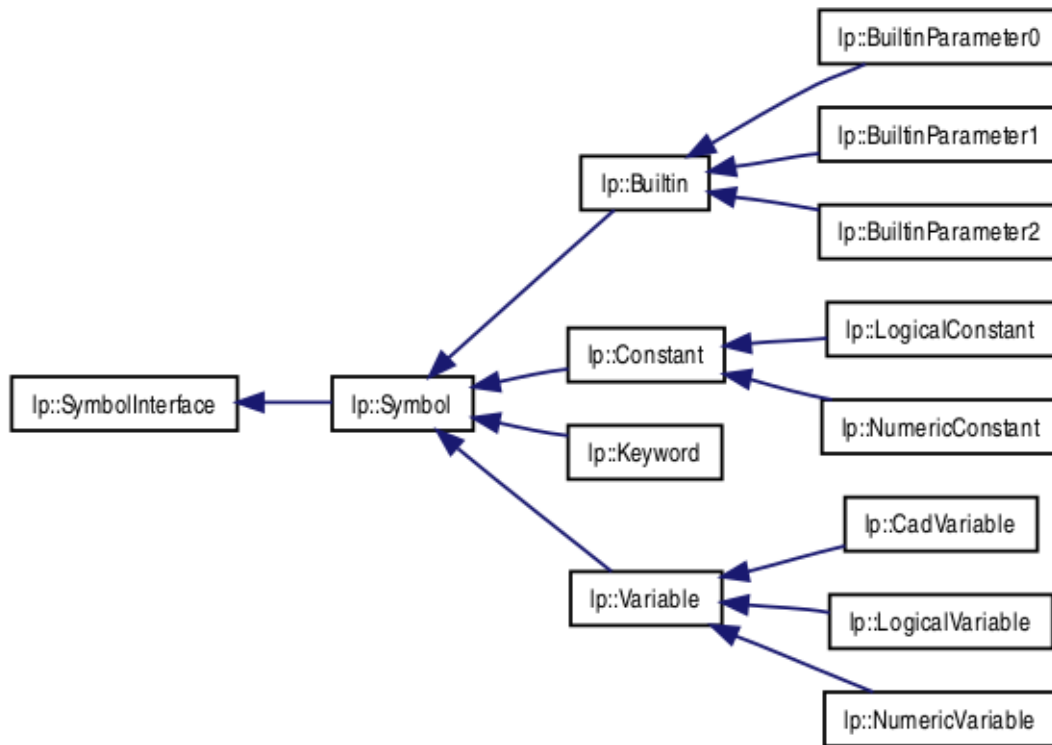
Ampliaciones

3 Tabla de símbolos

La tabla de símbolos está compuesta por los siguientes ficheros:

| | |
|---|--|
| ▼  table | |
|  builtin.cpp | Code of some functions of Builtin class |
|  builtin.hpp | Declaration of Builtin (built-in function) class |
|  builtinParameter0.cpp | Code of some functions of BuiltinParameter0 class |
|  builtinParameter0.hpp | Declaration of BuiltinParameter0 class |
|  builtinParameter1.cpp | Code of some functions of BuiltinParameter1 class |
|  builtinParameter1.hpp | Declaration of BuiltinParameter1 class |
|  builtinParameter2.cpp | Code of some functions of BuiltinParameter2 class |
|  builtinParameter2.hpp | Declaration of BuiltinParameter2 class |
|  CadVariable.cpp | Code of some functions of CadVariable class |
|  CadVariable.hpp | |
|  constant.cpp | Code of some functions of Constant class |
|  constant.hpp | Declaration of Constant class |
|  init.cpp | Code of the function for the initialization of table of symbols |
|  init.hpp | Prototype of the function for the initialization of table of symbols |
|  keyword.cpp | Code of some functions of Keyword class |
|  keyword.hpp | Declaration of Keyword class |
|  logicalConstant.cpp | Code of some functions of LogicalConstant class |
|  logicalConstant.hpp | Declaration of LogicalConstant class |
|  logicalVariable.cpp | Code of some functions of LogicalVariable class |
|  logicalVariable.hpp | Declaration of LogicalVariable class |
|  mathFunction.cpp | Code of mathematical functions |
|  mathFunction.hpp | Prototypes of mathematical functions |
|  numericConstant.cpp | Code of some functions of NumericConstant class |
|  numericConstant.hpp | Declaration of NumericConstant class |
|  numericVariable.cpp | Code of some functions of NumericVariable class |
|  numericVariable.hpp | Declaration of NumericVariable class |
|  symbol.cpp | Code of some functions of Symbol class |
|  symbol.hpp | Declaration of Symbol class |
|  symbolInterface.hpp | Declaration of abstract SymbolInterface class |
|  table.cpp | Code of some functions of Table class |
|  table.hpp | Declaration of TableInterface class |
|  tableInterface.hpp | Declaration of abstract TableInterface class |
|  variable.cpp | Code of some functions of Variable class |
|  variable.hpp | Declaration of Variable class |

La jerarquía de clases es del siguiente modo:



Para la implementación del intérprete, utilizamos todos los ficheros que se ven en la imagen de la parte superior. Cada fichero se encarga de introducir en la tabla de símbolos el símbolo correspondiente.

Cabe destacar que para la implementación de las cadenas, creamos unos nuevos ficheros de nombre "CadVariable.cpp" y "CadVariable.hpp" que contienen lo siguiente:

```

/#!/
    \file    CadVariable.hpp
    \brief   Declaration of CadVariable class
    \author
    \date    2017-12-1
    \version 1.0
*/

#ifndef _CADVARIABLE_HPP_
#define _CADVARIABLE_HPP_

#include <string>
#include <iostream>

#include "variable.hpp"

/#!/
    \namespace lp
    \brief Name space for the subject Language Processors
*/
namespace lp{

/#!/
    \class CadVariable

```

```

    \brief Definition of attributes and methods of CadVariable class
    \note CadVariable Class publicly inherits from Variable class
*/
class CadVariable:public lp::Variable
{
    /*!
    \name Private attributes of CadVariable class
    */
    private:
        std::string _value;    //!< \brief string value of the CadVariable

    /*!
    \name Public methods of CadVariable class
    */
    public:

    /*!
    \name Constructors
    */

    /*!
    \brief Constructor with arguments with default values
    \note Inline function that uses Variable's constructor as members initializer
    \param name: name of the CadVariable
    \param token: token of the CadVariable
    \param type: type of the CadVariable
    \param value: string value of the CadVariable
    \pre None
    \post A new CadVariable is created with the values of the parameters
    \sa setName, setValue
    */
    inline CadVariable(std::string name="", int token = 0, int type = 0,
std::string value=""): Variable(name,token,type)
    {
        this->setValue(value);
    }

    /*!
    \brief Copy constructor
    \note Inline function
    \param n: object of CadVariable class
    \pre None
    \post A new CadVariable is created with the values of an existent CadVariable
    \sa setName, setValue
    */
    CadVariable(const CadVariable & n)
    {
        // Inherited methods
        this->setName(n.getName());

        this->setToken(n.getToken());

        this->setType(n.getType());

        // Own method
        this->setValue(n.getValue());
    }

    /*!
    \name Observer
    */

```

```

/*!
    \brief Public method that returns the value of the CadVariable
    \note Función inline
    \pre None
    \post None
    \return Value of the CadVariable
    \sa getValue
*/
inline std::string getValue() const
{
    return this->_value;
}

/*!
    \name Modifier
*/

/*!
    \brief This functions modifies the value of the CadVariable
    \note Inline function
    \param value: new value of the CadVariable
    \pre None
    \post The value of the CadVariable is equal to the parameter
    \return void
    \sa setValue
*/
inline void setValue(const std::string & value)
{
    this->_value = value;
}

/*!
    \name I/O Functions
*/

/*!
    \brief Read a CadVariable
    \pre None
    \post The atributes of the CadVariable are modified with the read values
    \sa write
*/
void read();

/*!
    \brief Write a CadVariable
    \pre None
    \post None
    \sa read
*/
void write() const;

CadVariable &operator=(const CadVariable &n);

friend std::istream &operator>>(std::istream &i, CadVariable &n);

friend std::ostream &operator<<(std::ostream &o, CadVariable const &n);
*/

```



```

        \name Operators
    */
    //CadVariable &operator||(const CadVariable &n, const CadVariable &m);

// End of CadVariable class
};

// End of name space lp
}

// End of _CadVariable_HPP_
#endif

```

```

/#!
    \file    CadVariable.cpp
    \brief   Code of some functions of CadVariable class
    \author
    \date    2017-10-19
    \version 1.0
*/

#include <iostream>

// Delete the comment if you want to use atof in the operator overload >>
// #include <stdlib.h>

#include "CadVariable.hpp"

/*
Definitions of the read and write functions of the CadVariable class
*/

void lp::CadVariable::read()
{
    // Inherited attributes
    std::cout << "Name of the CadVariable: ";
    std::cin >> this->_name;

    std::cout << "Token of the CadVariable: ";
    std::cin >> this->_token;
    // The \n character is read
    std::cin.ignore();

    std::cout << "Type of the CadVariable: ";
    std::cin >> this->_type;
    // The \n character is read
    std::cin.ignore();

    // Own attribute
    std::cout << "Value of the CadVariable: ";
    std::cin >> this->_value;
    // The \n character is read
    std::cin.ignore();
}

```

```

void lp::CadVariable::write() const
{
    // Inherited methods
    std::cout << "Name:" << this->getName() << std::endl;
    std::cout << "Token:" << this->getToken() << std::endl;
    std::cout << "Type:" << this->getType() << std::endl;

    // Own method
    std::cout << "Value:" << this->getValue() << std::endl;
}

lp::CadVariable &lp::CadVariable::operator=(const lp::CadVariable &n)
{
    // Check that is not the current object
    if (this != &n)
    {
        // Inherited methods
        this->setName(n.getName());

        this->setToken(n.getToken());

        this->setType(n.getType());

        // Own method
        this->setValue(n.getValue());
    }

    // Return the current object
    return *this;
}

namespace lp{
    std::istream &operator>>(std::istream &i, lp::CadVariable &n)
    {
        // Inherited attributes
        i >> n._name;

        i >> n._token;
        // The \n character is read
        i.ignore();

        i >> n._type;
        // The \n character is read
        i.ignore();

        i >> n._token;
        // The \n character is read
        i.ignore();

        //////////////////////////////////////

        // Own attribute

        i >> n._value;
        // The \n character is read
        i.ignore();

        //////////////////////////////////////
    }
}

```

```

/* Alternative way using an auxiliar string

    std::string auxiliar;

    std::getline(i,auxiliar);
    n._token = atof(auxiliar.c_str());

    std::getline(i,auxiliar);
    n._type = atof(auxiliar.c_str());

    std::getline(i,auxiliar);
    n._value = atof(auxiliar.c_str());

*/

    // The input stream is returned
    return i;
}
std::ostream &operator<<(std::ostream &o, lp::CadVariable const &n)
{
    // Inherited attributes
    o << n._name << std::endl;

    o << n._token << std::endl;

    o << n._type << std::endl;

    // Own attribute
    o << n._value << std::endl;

    // The output stream is returned
    return o;
}
}
/*lp::CadVariable &lp::CadVariable::operator||(const lp::CadVariable &n, const
lp::CadVariable &m)
{
    // Check that is not the current object
    this = n + m;

    // Return the current object
    return *this;
}*/

```

Tanto la clase anterior como la clase “NumericVariable” derivan de la clase “variable” que es la encargada de introducir las variables a la tabla de símbolos. Esta clase es :

```

/*!
    \file    variable.hpp
    \brief   Declaration of Variable class
    \author
    \date    2017-12-1
    \version 1.0
*/

#ifdef _VARIABLE_HPP_

```

```

#define _VARIABLE_HPP_

#include <string>
#include <iostream>

#include "symbol.hpp"

/*!
    \namespace lp
    \brief Name space for the subject Language Processors
*/
namespace lp{

/*!
    \class      Variable
    \attention  Abstrac class
    \brief      Definition of atributes and methods of Variable class
    \note       Variable Class publicly inherits from Symbol class
*/
class Variable:public lp::Symbol
{
    /*!
    \name Protected atribute of Variable class
    */
    protected:
        int      _type;
    /*!< \brief Type of the  Variable: UNDEFINED, NUMERICVAR, STRINGVAR, LOGICALVAR

    */
    \name Public methods of Variable class
    */
    public:

    /*!
    \name Constructors
    */

    /*!
    \brief Constructor with arguments with default values
    \note  Inline function that uses Symbol's constructor as members initializer
    \param name: name of the Variable
    \param token: token of the Variable
    \param type: type of the Variable
    \pre   None
    \post  A new Variable is created with the values of the parameters
    \sa    setName, setToken, setType
    */

    inline Variable(std::string name="", int token = 0, int type = 0):
    Symbol(name,token)
    {
        this->setType(type);
    }

    /*!
    \brief Copy constructor
    \note  Inline function
    \param s: object of Variable class
    \pre   None
    \post  A new Variable is created with the values of an existent Variable
    \sa    setName, setValue
    */

    inline Variable(const Variable & s)

```

```

    {
        // Inherited methods
        this->setName(s.getName());
        this->setToken(s.getToken());

        // Own method
        this->setType(s.getType());
    }

    /*!
    \name Observer
    */

    /*!
    \brief    Public method that returns the type of the Variable
    \note     Inline function
    \pre      None
    \post     None
    \return   int Type of the Variable
    \sa       getName, getToken
    */
    inline int getType() const
    {
        return this->_type;
    }

    /*!
    \name Modifier
    */

    /*!
    \brief    This functions modifies the token of the Variable
    \note     Función inline
    \param    type: new type of the Variable
    \pre      None
    \post     The type of the Variable is equal to the parameter
    \return   void
    \sa       setName, setToken
    */
    inline void setType(int type)
    {
        this->_type = type;
    }

    /*!
    \name Operator
    */

    /*!
    \brief    Assignment Operator
    \note     Virtual method: can be redefined in the heir class
    \param    v: objecto of Variable class
    \post     The atributes of this object are equal to the atributes of the
parameter
    \return   Reference to this object
    */
    virtual Variable &operator=(const Variable &v);

```

```

/*!
    \name I/O Functions
*/

/*!
    \brief Write a Variable
    \note Virtual method: can be redefined in the heir class
    \pre None
    \post None
    \sa read
*/
    virtual void write() const;

/*!
    \brief Read a Variable
    \note Virtual method: can be redefined in the heir class
    \pre None
    \post The attributes of the Variable are modified with the read values
    \sa writeS
*/
    virtual void read();

// End of Variable class
};

// End of name space lp
}

// End of _VARIABLE_HPP_
#endif

```

```

/*!
    \file variable.cpp
    \brief Code of some functions of Variable class
    \author
    \date 2017-10-19
    \version 1.0
*/

#include <iostream>
#include "variable.hpp"

/*
    Operator
*/

lp::Variable &lp::Variable::operator=(const lp::Variable &v)
{
    // Check that is not the current object
    if (this != &v)
    {

```

```

        // Inherited methods
        this->setName(v.getName());

        this->setToken(v.getToken());

        // Own method
        this->setType(v.getType());
    }

    // Return the current object
    return *this;
}

/*
    I/O Functions
*/

void lp::Variable::read()
{
    // Inherited attributes
    std::cout << "Name of the Variable: ";
    std::cin >> this->_name;

    std::cout << "Token of the Variable: ";
    std::cin >> this->_token;
    // The \n character is read
    std::cin.ignore();

    // Own attribute
    std::cout << "Type of the Variable: ";
    std::cin >> this->_type;
    // The \n character is read
    std::cin.ignore();
}

void lp::Variable::write() const
{
    // Inherited methods
    std::cout << "Name:" << this->getName() << std::endl;
    std::cout << "Token:" << this->getToken() << std::endl;

    // Own method
    std::cout << "Token:" << this->getType() << std::endl;
}

```

4 Análisis léxico

Los componentes léxicos que tenemos en nuestro intérprete son los siguientes:

```
DIGIT    [0-9]
LETTER   [a-zA-Z]
NUMBER   {DIGIT}+(\.{DIGIT}+)?([Ee][+|-]?{DIGIT}+)?
IDENTIFIER {LETTER}({LETTER}|{DIGIT}|"_"({LETTER}|{DIGIT}))*
ERRORID1 {IDENTIFIER}"_"
ERRORID2 {DIGIT}{IDENTIFIER}
ERRORID3 "_" {IDENTIFIER}
ERRORID4 {LETTER}({LETTER}|{DIGIT}|"_"({LETTER}|{DIGIT}))*
SIMBOL   ("$"|"&")
CADENA   "'"([^\']|"\\\'")*"'
COMENTARIO_VARIAS <<([^(>)]*)>>
COMENTARIO_UNA    @([^\n])*
```

- **Dígito:** números del 0 al 9.
- **Letras:** letras de la “a” a la “z” tanto en mayúscula como en minúscula.
- **Números:** deben empezar con un número seguido de otro número o un punto en caso de números decimales o de una “e” para notación científica.
- **Identificador:** los identificadores empiezan con una letra seguido de un número, letra o barra baja seguido de un número o letra. Recordemos que no puede haber dos barras bajas seguidas.
- **Errores:** a continuación tenemos una serie de detección de errores para los identificadores incorrectos. Estos son los que acaban en “_”, los que empiezan por algo distinto de una letra y para los que tienen dos barras bajas seguidas.
- **Símbolos prohibidos:** en este caso los símbolos prohibidos son “\$” y “&”. Cuando se utilice uno de estos saldrá un mensaje de error.
- **Comentarios:** los comentarios de una línea se reconocen con el símbolo @. Y los comentarios de varias líneas se reconocen con los símbolos “<<” y “>>”, el primero para abrir el comentario y el segundo para cerrarlo.

Cuando detecta un identificador, el intérprete realiza lo siguiente:


```

{IDENTIFIER}
{
    for(int i = 0; yytext[i]!='\0'; i++){
        yytext[i] = tolower(yytext[i]);
    }
    /* NEW in example 7 */
    std::string identifier(yytext);

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

    /* If the identifier is not in the table of symbols then it is inserted */
    if (table.lookupSymbol(identifier) == false){
    /*
    The identifier is inserted into the symbol table as undefined Variable with value
    0.0
    */
        lp::NumericVariable *n = new
        lp::NumericVariable(identifier, VARIABLE, UNDEFINED, 0.0);

    /* A pointer to the new NumericVariable is inserted into the table of symbols */
        table.installSymbol(n);
        return VARIABLE;
    }

    /* 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();
    }
}

```

Como nos pide que no diferencie entre mayúsculas y minúsculas, transforma al identificador en minúsculas. Primero comprueba que el identificador que tenemos está insertado en la tabla de símbolos, si es así, devuelve el token. En caso contrario, mete el identificador en la tabla de símbolos con valor indefinido y se inserta un puntero a la nueva variable insertada en la tabla de símbolos.

Con los números hace lo siguiente:

```

{NUMBER}

```

```

{
    /* MODIFIED in example 4 */
    /* Conversion of type and sending of the numerical value to the parser
*/
    yylval.number = atof(yytext);

    return NUMBER;
}

```

El valor recibido se transforma a flotante y devuelve NUMBER.

Con una cadena:

```

{CADENA}    { yylval.cadena= yytext;
              return CADENA;}

```

Ahora vamos a poner de forma general los diferentes casos en los que nuestro intérprete detecta según las normas del pseudocódigo:

```

{COMENTARIO_VARIAS} {printf("Encontrado comentario de varias lineas\n") ;}

{COMENTARIO_UNA}    {printf("Encontrado comentario de una linea\n");}

{ERRORID1} {printf("Error en el identificador %s.\nEl identificador no puede acabar
en '_' \n",yytext);}

{ERRORID2} {printf("Error en el identificador %s.\nEl identificador no puede empezar
en numero.\n",yytext);}

{ERRORID3} {printf("Error en el identificador %s. \nEl identificador no puede
empezar en '_' \n",yytext);}

{ERRORID4} {printf("Error en el identificador %s\nEl identificador no puede contener
dos '_' seguidas.\n",yytext);}

{SIMBOL}    {printf("Simbolo no permitido %s\n", yytext);}

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

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

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

"#mod"       { return MODULO; }          /* NEW in example 5 */

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

"**"         {return POTENCIA;}

```

```

"^"    {    printf("Error: Quizas queria poner '**'\n");
           return 0; }    /* NEW in example 15 */

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

"=="   {    printf("Error: Quizas queria poner '='\n");
           return 0; }    /* NEW in example 15 */

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

"><"   {    printf("Error: Quizas queria poner '<'\n");
           return 0; }    /* NEW in example 15 */

"!="    {    printf("Error: Quizas queria poner '<'\n");
           return 0; }    /* NEW in example 15 */

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

">"    {    printf("Error: Quizas queria poner '>'\n");
           return 0; }    /* NEW in example 15 */

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

"<"    {    printf("Error: Quizas queria poner '<'\n");
           return 0; }    /* NEW in example 15 */

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

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

"#no"   { return NOT; }    /* NEW in example 15 */

"#o"    { return OR; }    /* NEW in example 15 */

"#y"    { return AND; }    /* NEW in example 15 */

"&&"   {    printf("Error: Quizas queria poner '#y'\n");
           return 0; }    /* NEW in example 15 */

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

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

"#borrar" { return BORRAR; }

"#lugar" {return LUGAR;}

"||"    {return CONCAT;}

```

Cada vez que detecta un símbolo, hace un return del símbolo encontrado. Estos símbolos se desarrollan en el fichero “interpreter.y” en el que se controla cada símbolo y le da un sentido para la gramática.

En caso de encontrar un comentario, simplemente da un aviso de que lo ha encontrado.

Además tenemos implementados los métodos “#borrar”, que consiste en liberar el espacio que hay en pantalla; y “#lugar” que nos mueve al lugar que queramos.

5 Análisis sintáctico

El análisis léxico se desarrolla en el fichero “interprete.y”. Los símbolos de la gramática son:

- **Símbolos terminales:** los símbolos terminales en nuestro intérprete están declarado del siguiente modo:

```
%token SEMICOLON
/*****/

/* NEW in example 17: IF, ELSE, WHILE */
%token PRINT READ IF THEN ELSE END_IF WHILE DO END_WHILE REPETIR UNTIL FOR FROM PASS
END_FOR CASE VALUE END_CASE LUGAR CASOS VALOR DEFECTO FIN_CASOS BORRAR COLON
PRINTCAD READ_CAD

/* NEW in example 17 */
%token LETFCURLYBRACKET RIGHTCURLYBRACKET
/* NEW in example 7 */
%right ASSIGNMENT

/* NEW in example 14 */
%token COMMA

/*****/
/* MODIFIED in example 4 */
%token <number> NUMBER
/*****/
%token <cadena> CADENA
/*****/
/* NEW in example 15 */
%token <logic> BOOL
/*****/

/* MODIFIED in examples 11, 13 */
%token <identifier> VARIABLE UNDEFINED CONSTANT BUILTIN

/* Left associativity */

/*****/
/* NEW in example 15 */
%left OR

%left AND

%nonassoc GREATER_OR_EQUAL LESS_OR_EQUAL GREATER_THAN LESS_THAN EQUAL NOT_EQUAL

%left NOT
/*****/

/* MODIFIED in example 3 */
%left PLUS MINUS CONCAT

/* MODIFIED in example 5 */
%left MULTIPLICATION DIVISION MODULO DIVENTERA

%left POTENCIA
```

```
%left LPAREN RPAREN

%nonassoc UNARY

// Maximum precedence
/* MODIFIED in example 5 */
%right POWER
```

- Símbolos no terminales: los símbolos no terminales son:

```
%type <expNode> exp cond

/* New in example 14 */
%type <parameters> listOfExp restOfListOfExp

%type <stmts> stmtlist

// New in example 17: if, while, block
%type <st> stmt asgn print read if while repetir para //block

%type <prog> program

%type <casos> valores
```

La estructura unión tiene la siguiente forma:

```
%union {

    char * identifier;                /* NEW in example 7 */
    double number;
    bool logic;                       /* NEW in example 15 */
    char * cadena;
    lp::ExpNode *expNode;             /* NEW in example 16 */

    std::list<lp::ExpNode *> *parameters;
    // New in example 16; NOTE: #include<list> must be in interpreter.l, init.cpp,
    interpreter.cpp

    std::list<lp::Statement *> *stmts; /* NEW in example 16 */
    lp::Statement *st;                 /* NEW in example 16 */
    lp::AST *prog;                     /* NEW in example 16 */
    lp::CasosStmt *casos;

}
```

Ahora veamos cómo se genera la gramática en nuestro intérprete. Todo esto se declara en el fichero “intérprete.y”. El intérprete, una vez recibida una sentencia y tras hacer el análisis léxico estudia sus sintaxis la cual veremos a continuación:

En primer lugar crea un nuevo AST con el siguiente código:

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

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

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

Creamos una lista de stmt para poder hacer varias acciones durante la ejecución del intérprete. Esto se utilizará en las sentencias de control, en las cuales les podremos asignar que hagan varias acciones durante su funcionamiento.

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

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

    // Control the interactive mode of execution of the interpreter
    if (interactiveMode == true && control == 0)
    {
        for(std::list<lp::Statement *>::iterator it = $$->begin();
            it != $$->end();
            it++)
        {
            (*it)->print();

            (*it)->evaluate();
        }

        // Delete the AST code, because it has already run in the interactive mode.
        $$->clear();
    }
}

| stmtlist error
{
    // just copy up the stmtlist when an error occurs
    $$ = $1;
```

```
// The previous look-ahead token ought to be discarded with `yyclearin;`
    yyclearin;
}
;
```

Creamos un stmt para el final de cada sentencia. En nuestro caso, el final será SEMICOLON que corresponde al punto y coma. Esto hará que cuando escribamos punto y coma al final de una sentencia, detecte que está al final y finalice.

```
stmt: SEMICOLON /* Empty statement: ";" */
{
    // Create a new empty statement node
    $$ = new lp::EmptyStmt();
}
| asgn SEMICOLON
{
    // Default action
    // $$ = $1;
}
| exp SEMICOLON
{
}
| LUGAR LPAREN exp COMMA exp RPAREN SEMICOLON
{
    $$ = new lp::LugarStmt($3, $5);
}
| print SEMICOLON
{
    // Default action
    // $$ = $1;
}
| read SEMICOLON
{
    // Default action
    // $$ = $1;
}
/* NEW in example 17 */
| borrar SEMICOLON
{
}
| lugar SEMICOLON
{
}
| if SEMICOLON
{
    // Default action
```

```

        // $$ = $1;
    }
    /* NEW in example 17 */
    | while SEMICOLON
    {
        // Default action
        // $$ = $1;
    }
;
    //NEW in example 17
    | block
    {
        // Default action
        // $$ = $1;
    }

    | repetir SEMICOLON
    {

    }

    | para SEMICOLON
    {

    }

    | casos SEMICOLON
;

```

Necesitamos un sistema que nos ayude a controlar el modo interactivo, para ello utilizamos el controlsymbol:

```

controlSymbol: /* Epsilon rule*/
{
    control++;
}
;

```

IF

Ahora veamos las sentencias de control IF. Esta se divide en dos, una parte para el “if” que no tenga “else” y otra que si lo contenga. El if tiene que tener el token de IF que se corresponde con “si”, una condición, el token THEN que se corresponde con “entonces”, una lista de sentencias y el token END_IF que es “fin_si”. En caso de que utilicemos “else”, entre THEN y END_IF deberemos poner el token ELSE que se corresponde a “si_no” y una lista de sentencias. Las listas de sentencias son las acciones que queramos que realice el programa.

Cuando esto sea detectado por el intérprete, se creará un nuevo IfStmt donde se comprobará y realizará las acciones correspondientes con el if y el else, como comprobar la condición y realizar la lista de sentencias.

```

if: /* Simple conditional statement */
    IF controlSymbol cond THEN stmtlist END_IF

```



```

{
    // Create a new if statement node
    $$ = new lp::IfStmt($3, $5);

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

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

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

```

Veamos un ejemplo muy sencillo de cómo sería el if:

```

dato:=4;

si (dato > 5) entonces
    escribir_cadena('Dato es mayor que 5');
    escribir(dato);

si_no
    escribir_cadena('Dato no es mayor que 5');
    escribir(dato);
fin_si;

```

Como dato no es mayor que 5, nos iríamos al else y se ejecutan esas sentencias.

Las palabras señaladas en color rojo se corresponden con los tokens IF, THEN, ELSE y END_IF, el color azul señala la condición y por último, el color verde nos indica la lista de sentencias que se desea realizar.

WHILE

El bucle while necesita el token WHILE que se corresponde con “mientras”, una condición, el token DO, que se corresponde con “hacer”, una lista de sentencias y el token END_WHILE que se corresponde con “fin_mientras”.

```

while: WHILE controlSymbol cond DO stmtlist END_WHILE
      {
          // Create a new while statement node

```

```

        $$ = new lp::WhileStmt($3, $5);

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

```

Un ejemplo del uso del while sería:

```

dato:=0;

mientras (dato<10) hacer
    escribir(dato);
    dato:=dato+1;
fin_mientras;

```

La salida sería una cuenta del 0 al 9.

Las palabras señaladas en color rojo se corresponden con los tokens WHILE, DO y END_WHILE, el color azul señala la condición y por último, el color verde nos indica la lista de sentencias que se desea realizar.

REPETIR

El bucle repetir se compone del token REPETIR, que se corresponde con “repetir”; una lista de sentencias, el token UNTIL que se corresponde con “hasta” y una condición.

```

repetir: REPETIR stmtlist UNTIL controlSymbol cond
    {
        $$ = new lp::RepetirStmt($2, $5);
        control--;
    }
;

```

Veamos un ejemplo de esta sentencia de control:

```

dato:=0;

repetir
    escribir(dato);
    dato:=dato+1;
hasta(dato=10);

```

La salida es igual que el ejemplo anterior, los números del 0 al 9.

Las palabras señaladas en color rojo se corresponden con los tokens REPETIR, y UNTIL, el color azul señala la condición y por último, el color verde nos indica la lista de sentencias que se desea realizar.

PARA

El bucle para es el más complejo de todos puesto que se divide en dos. La opción pass es opcional así que hemos creado dos casos. Uno en que esté presente y otra en que no esté.

En caso que no esté presente necesitamos el token FOR que se corresponde con “para”, una variable(esta variable se comprueba si existe o no. Se explica mejor en el punto 6), el token FROM que se corresponde con “desde”, una expresión, la cual nos marcará el inicio del bucle, el token UNTIL que se corresponde con “hasta”, una expresión que será hasta donde llegue el bucle, el token DO que se corresponde con “hacer”, una lista de sentencias y por último el token END_FOR, que se corresponde con “fin_para”.

Si está presente lo tenemos que poner entre UNTIL y DO. Utilizamos el token PASS, que se corresponde con paso y una expresión que será cuánto aumentará el iterador del bucle con cada paso.

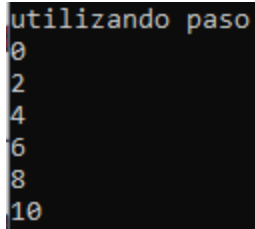
```
para:
  FOR controlSymbol VARIABLE FROM exp UNTIL exp DO stmtlist END_FOR
  {
    $$ = new lp::ForStmt($3, $5, $7, $9);
    control--;
  }
| FOR controlSymbol VARIABLE FROM exp UNTIL exp PASS exp DO stmtlist END_FOR
{
  $$ = new lp::ForStmt($3, $5, $7, $9, $11);
  control--;
}
;
```

A continuación mostraremos dos ejemplos, uno en que se utilice paso y otro en que no:

```
escribir_cadena('Utilizando paso\n');

para x desde 0 hasta 10 paso 2 hacer
    escribir(x);
fin_para
```

La salida mostraría los números pares del 0 al 10. La siguiente imagen lo demuestra:



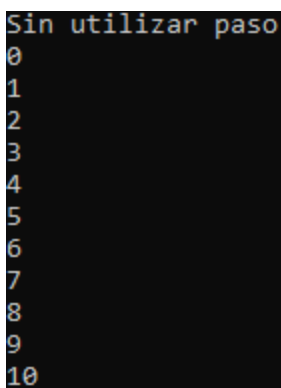
```
utilizando paso
0
2
4
6
8
10
```

Las palabras señaladas en color rojo se corresponden con los tokens FOR, y FROM, UNTIL, PASS, DO y END_FOR y el color verde nos indica la lista de sentencias que se desea realizar.

En caso que no utilizemos la variable paso, el valor por defecto de esta variable es 1, por lo tanto la cuenta ascenderá de uno en uno.

```
escribir_cadena('Sin utilizar paso\n');
para x desde 0 hasta 10 hacer
    escribir(x);
fin_para
```

La salida mostraría los números del 0 al 10. La siguiente imagen lo demuestra:



```
Sin utilizar paso
0
1
2
3
4
5
6
7
8
9
10
```

SWITCH

Para hacer esta sentencia, la dividimos en dos partes, una que controle el switch y otra que vaya cogiendo los valores de cada parte del switch. A su vez el último apartado está dividido en dos para controlar el valor de defecto. Esta función es recursiva como se puede observar en el código, así, le podremos ir añadiendo valores al switch hasta el último que será el valor defecto.

En primer lugar el switch recibe el token CASOS que se corresponde con “casos”, a continuación una expresión, valores, que esto se mandará a la siguiente función y FIN_CASOS que se corresponde con “fin_casos”.

En el caso de valores recibirá VALOR que se corresponde con “valor”, una expresión que será el número que utiliza la opción del switch, COLON que se corresponde con dos puntos, una lista de acciones y volverá a recibir valores para hacer otro caso de switch.

Por último si utilizamos defecto, necesitamos el token DEFECTO que se corresponde con “defecto”, dos puntos y una lista de acciones. Tras esto finalizará el switch volviendo a casos y utilizando FIN_CASOS como se explicó con anterioridad.

```
casos:
    CASOS controlSymbol exp valores FIN_CASOS
    {
        $$ = new lp::SwitchStmt($3,$4);
        control --;
    }

;
valores: VALOR controlSymbol exp COLON stmtlist valores
    {
        $$ = new lp::CasosStmt($3,$5,$6);
        control--;
    }
    | DEFECTO controlSymbol COLON stmtlist
    {
        $$ = new lp::CasosStmt($4);
        control--;
    }

;
```

Veamos un ejemplo muy sencillo de la utilización del switch. El pseudocódigo será el siguiente:

```

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

casos(dato)
    valor 1:
        escribir_cadena('opcion 1\n');
    valor 2:
        escribir_cadena('opcion 2\n');
    defecto:
        escribir_cadena('opcion defecto\n');
fin_casos;

```

El color rojo representa los tokens, el verde representa las acciones que va a realizar y el azul el valor que debe tomar la expresión para que entre en esa parte del switch.

El programa nos pedirá que insertemos un número, si insertamos un 1, irá a la primera parte del switch, si insertamos un 2, iremos a la segunda parte y si insertamos cualquier otro número nos iremos a la parte de defecto

```

    inserte un nmero --> 1
opcion 1

```

```

    inserte un nmero --> 2
opcion 2

```

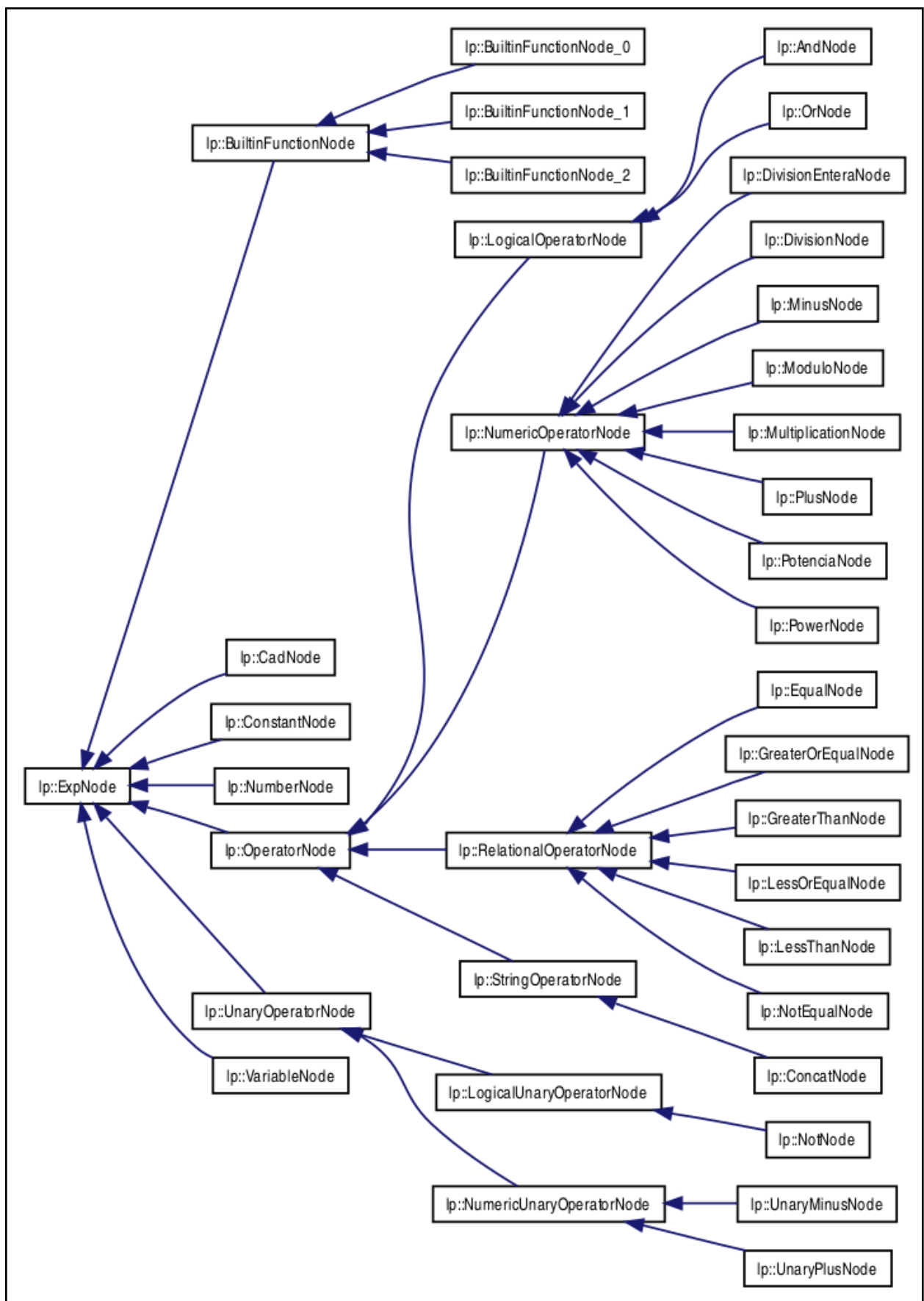
```

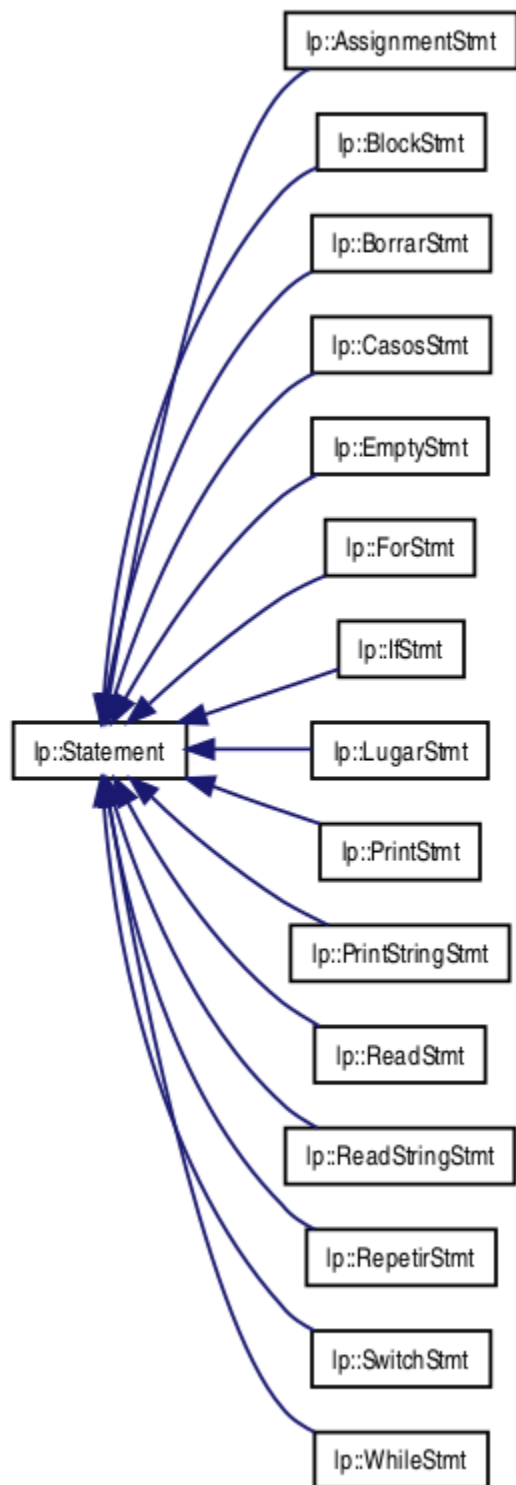
    inserte un nmero --> 3
opcion defecto

```

6 Código de AST

Las clases de AST están organizadas de la siguiente manera:





Como se puede ver la clase ExpNode es heredada por todas las demás.
Ahora vamos a ver todas las clases que tenemos en el código ASP.

```
class ExpNode
{
public:
    /*!
        \brief   Type of the expression
        \warning Pure virtual function: must be redefined in the heir classes
        \return  int
        \sa      print
    */
    virtual int getType() = 0;

    /*!
        \brief   Print the expression
        \warning Pure virtual function: must be redefined in the heir classes
        \return  void
        \sa      evaluate()
    */
    virtual void print() = 0;

    /*!
        \brief   Evaluate the expression as NUMBER
        \warning Virtual function: could be redefined in the heir classes
        \return  double
        \sa      print
    */
    virtual double evaluateNumber()
    {
        return 0.0;
    }

    /*!
        \brief   Evaluate the expression as BOOL
        \warning Virtual function: could be redefined in the heir classes
        \return  bool
        \sa      print
    */
    virtual bool evaluateBool()
    {
        return false;
    }

    virtual std::string evaluateCad()
    {
        return "";
    }
};
```

Para implementar las cadenas, está añadido el método string evaluateCad().

A continuación tenemos la clase “VariableNode” que es la que se encarga del control de las variables numéricas:

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

    public:

        /*!
            \brief Constructor of VariableNode
            \param value: double
            \post A new NumericVariableNode 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();

        std::string evaluateCad();
};
```

Para el control de las variables de tipo cadena tenemos la siguiente clase:

```

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

    public:

        /*!
            \brief Constructor of CadNode
            \param value: string
            \post A new CadVariableNode is created with the name of the parameter
            \note Inline function
        */
        CadNode(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 string
            \sa print
        */
        std::string evaluateCad();
};

```

Para los valores numéricos existe la siguiente clase:

```

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();
};

```

Para los operadores unarios tenemos las tres siguientes clases:

```

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

public:

    /*!
        \brief Constructor of UnaryOperatorNode links the node to its 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();
        }
};

class NumericUnaryOperatorNode : public UnaryOperatorNode
{
public:

    /*!
        \brief Constructor of NumericUnaryOperatorNode uses UnaryOperatorNode's
        constructor as member initializer
        \param expression: pointer to ExpNode
        \post A new NumericUnaryOperatorNode is created with the parameters
        \note Inline function
    */
    NumericUnaryOperatorNode(ExpNode *expression): UnaryOperatorNode(expression)
    {
        // Empty
    }

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

class LogicalUnaryOperatorNode : public UnaryOperatorNode
{
public:

    /*!
        \brief Constructor of LogicalUnaryOperatorNode uses UnaryOperatorNode's
        constructor as member initializer
        \param expression: pointer to ExpNode
        \post A new NumericUnaryOperatorNode is created with the parameters
        \note Inline function
    */
    LogicalUnaryOperatorNode(ExpNode *expression): UnaryOperatorNode(expression)
    {
        // Empty
    }

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

```

Para los operadores unarios tenemos las siguientes clases, una con el símbolo + y otra con el símbolo menos:

```

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 as
        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();
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    /*!
        \class    UnaryPlusNode
        \brief    Definition of attributes and methods of UnaryPlusNode class
        \note     UnaryPlusNode Class publicly inherits from NumericUnaryOperatorNode class
    */
class UnaryPlusNode : public NumericUnaryOperatorNode
{
public:
    /*!
        \brief Constructor of UnaryPlusNode uses NumericUnaryOperatorNode's
        constructor as member initializer
        \param expression: pointer to ExpNode
        \post  A new UnaryPlusNode is created with the parameter
    */
    UnaryPlusNode(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();
};

```

Tenemos una clase para controlar las operaciones:

```

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;
    }
};

```

Tenemos dos clases de operadores para números y para cadenas:

```

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();
};

class ConcatNode : public StringOperatorNode
{
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
    */
    ConcatNode(ExpNode *L, ExpNode *R) : StringOperatorNode(L,R)
    {
        // Empty
    }

    void print();

    std::string evaluateCad();
};

```

Ahora, tenemos una clase para cada operación. Ponemos el de restar como ejemplo porque el resto son muy parecidos:

```

class MinusNode : public NumericOperatorNode
{
public:

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

```



```

*/
void print();

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

```

También tenemos clases para los operadores de asignación:

```

class GreaterThanNode : public RelationalOperatorNode
{
public:

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

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

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

```

Las clases para los operadores lógicos:

```

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();
};

```

La clase statement

```

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;
};

```

La clase assignment se encarga de asignar valores a las variables.

```

class AssignmentStmt : public Statement
{

```

```

private:
    std::string _id;    //!< Name of the variable of the assignment statement
    ExpNode *_exp;      //!< Expression the assignment statement

    AssignmentStmt *_asgn; //!< Allow multiple assignment -> 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 assignment -> 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();
};

```

La clase PrintStmt hace funcionar la parte de imprimir por pantalla del intérprete:

```
class PrintStmt: public Statement
{
private:
    ExpNode *_exp; //!< Expression the print statement

public:
    /*!
        \brief Constructor of PrintStmt
        \param expression: pointer to ExpNode
        \post A new PrintStmt is created with the parameter
    */
    PrintStmt(ExpNode *expression)
    {
        this->_exp = expression;
    }

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

    /*!
        \brief Evaluate the PrintStmt
        \return double
        \sa print
    */
    void evaluate();
};
```

La siguiente clase se encarga de leer las variables cuando usemos el comando “leer”

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

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

    /*!
        \brief Print the ReadStmt
        \return void
        \sa evaluate()
    */
};
```

```

    void print();

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

class ReadStringStmt : public Statement
{
    private:
        std::string _id; //!< Name of the ReadStmt

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

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

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

```

Tenemos las clases de los diferentes bucles y condicionales del intérprete:

```

class IfStmt : public Statement
{
    private:
        ExpNode *_cond; //!< Condicion of the if statement
        Statement *_stmt1; //!< Statement of the consequent
        Statement *_stmt2; //!< Statement of the alternative

    public:
    /*!
        \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, Statement *statement1, Statement *statement2)
{
    this->_cond = condition;
    this->_stmt1 = statement1;
    this->_stmt2 = statement2;
}

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

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

/*!
    \class WhileStmt
    \brief Definition of atributes and methods of WhileStmt class
    \note WhileStmt Class publicly inherits from Statement class
           and adds its own print and evaluate functions
*/

class WhileStmt : public Statement
{
private:
    ExpNode *_cond; //!< Condicion of the while statement
    Statement *_stmt; //!< Statement of the body of the while loop

public:
    /*!
        \brief Constructor of WhileStmt
        \param condition: ExpNode of the condition
        \param statement: Statement of the body of the loop
        \post A new WhileStmt is created with the parameters
    */
    WhileStmt(ExpNode *condition, Statement *statement)

```

```

        {
            this->_cond = condition;
            this->_stmt = statement;
        }

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

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

class RepetirStmt : public Statement
{
private:
    ExpNode *_cond;
    Statement *_stmt;

public:

    RepetirStmt(Statement *statement, ExpNode *condition)
    {
        this->_cond = condition;
        this->_stmt = statement;
    }

    void print();

    void evaluate();
};

class ForStmt : public Statement
{
private:
    std::string _id;
    ExpNode *_from;

```

```

ExpNode *_until;
ExpNode *_pass;

std::list<Statement *> *_stmt; //!< Statement of the consequent

public:

ForStmt(std::string id, ExpNode *from, ExpNode *until, std::list<Statement*> *statement)
{
    this->_id = id;
    this->_from = from;
    this->_until = until;
    this->_stmt = statement;
}

ForStmt(std::string id, ExpNode *from, ExpNode *until, ExpNode *pass, std::list<Statement*>
*statement)
{
    this->_id = id;
    this->_from = from;
    this->_until = until;
    this->_pass = pass;
    this->_stmt = statement;
}

void print();

void evaluate();
};

class CasosStmt : public Statement
{
private:

ExpNode *_exp;
std::list<Statement *> *_stmts;
CasosStmt *_valores;
int _type;

```



```
public:
```

```
CasosStmt(ExpNode * exp, std::list<Statement *> *stmts, CasosStmt *valores)
```

```
{  
    this->_exp = exp;  
    this->_stmts = stmts;  
    this->_valores = valores;  
    _type = 0;  
}
```

```
CasosStmt(std::list<Statement *> *stmts)  
{
```

```
    this->_stmts = stmts;  
    _type = 1;  
}
```

```
void print();
```

```
void evaluate(double valor);
```

```
void evaluate(){}  

```

```
double getValor(){  
    return this->_exp->evaluateNumber();  
}
```

```
};  
////////////////////////////////////////////////////////////////
```

```
class SwitchStmt : public Statement
```

```
{  
private:
```

```
    ExpNode *_exp;  
    CasosStmt *_valores;
```

```
public:

SwitchStmt(ExpNode * exp,CasosStmt *valores)
{
    this->_exp = exp;
    this->_valores = valores;
}

void print();

void evaluate();

};
```

Por último tenemos la clase borrar que se encarga de controlar el comando “#borrar”

```
class BorrarStmt : public Statement
{
private:

public:

BorrarStmt()
{
}

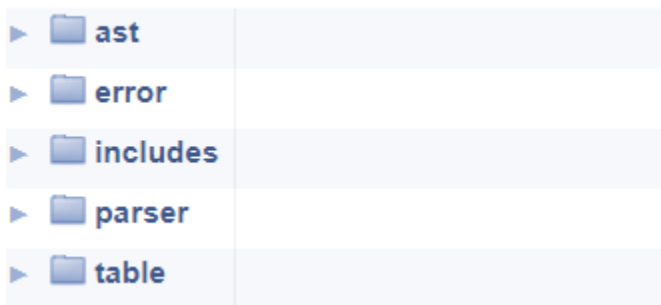
void print();

void evaluate();

};
```

7 Modo de obtención del intérprete

En primer lugar, tenemos el fichero principal en el que se encuentran los ficheros principales del intérprete como el makefile y el .cpp. También tenemos el ejecutable y un makefile para la opción de crear documentación con doxygen. Con esta documentación generada por doxygen, podemos ver los diferentes directorios que tenemos y sus diferentes ficheros:



AST

En él se encuentra la clase AST que está explicada en el [punto 6](#). Dentro de este directorio se encuentra ast.cpp, ast.hpp y el makefile. Estos se encargan de crear las clases y las funciones del árbol AST.

Error

Aquí está el fichero error con la que se controlan los errores que puedan ocurrir en el intérprete. Los ficheros que se encuentran aquí son: error.cpp, error.hpp y el makefile.

Includes

Se encuentra el fichero macros.hpp en el que están definidas las macros del intérprete como por ejemplo PLACE o los diferentes colores. Este fichero se utiliza para los comandos #lugar y #borrar.

Parser

En este fichero están los ficheros de [flex](#) y [yacc](#) del intérprete.

Table

Aquí se encuentran todos los ficheros relacionados con la tabla de símbolos. En este directorio, hemos creado el fichero CadVariable para introducir las cadenas en la tabla de símbolos.

8 Modo de ejecución del intérprete

Interactiva

Para ejecutar el intérprete en modo iterativo, basta con ejecutar el programa desde la terminal sin ningún argumento. Esto se controla desde el fichero “ipe.cpp” en el que tenemos la variable “interactiveMode”, la cual, en este caso será true.

Veamos un sencillo ejemplo de ejecución en modo interactivo. Vamo a hacer un pseudocódigo que pida dos números al usuario y los multiplique:

1. En primer lugar, ejecutamos el programa en la terminal con “./ipe.exe”



```
i82carij@NEWTS:~/3º/PL/P2/V1$ ./ipe.exe
```

2. Una vez dentro del intérprete podemos utilizarlo

A partir de un fichero

Para ejecutarlo a partir de un fichero habría que ejecutar el intérprete desde la terminal añadiendo como parámetro el nombre del fichero que queremos leer. Esto pondrá la variable que controla el modo a false.

Cabe destacar que en el código se controla si el fichero introducido existe o no y si su extensión es correcta. Esto lo hace de la siguiente forma:

```
if (argc == 2)
{
    std::string nombre = argv[1];
    yyin = fopen(argv[1], "r");

    if (yyin == NULL)
    {
        printf("El fichero no existe\n");
        return 0;
    }
    else{
        if(nombre.substr(nombre.find_last_of(".") + 1) != "e") {
            printf("La extension del fichero no es valida\n");
        }
    }

    interactiveMode = false;
}
```

Veámos un ejemplo de cómo se ejecuta:

```
i82carij@NEWS:~/3º/PL/P2/V1$ ./ipe.exe menu.e
```

Con esto ya se estaría ejecutando el fichero. En nuestro intérprete hemos implementado un control de errores para los ficheros. Se controla que la extensión del archivo sea “.e” en caso contrario mandaría un mensaje de error y también se comprueba que el fichero exista. Veamos un par de ejemplos de esto.

| | |
|---|--|
| <pre>i82carij@NEWS:~/3º/PL/P2/V1\$./ipe.exe fichero.e El fichero no existe</pre> | <pre>i82carij@NEWS:~/3º/PL/P2/V1\$./ipe.exe fichero.txt La extension del fichero no es valida</pre> |
| El fichero no existe | No tiene la extensión correcta |

9 Ejemplos

Veamos primero los ejemplos que se nos dan en la práctica:

Conversion.e

```
<<
  Asignatura:      Procesadores de Lenguajes

  Titulaci3n:      Ingeniería Informática
  Especialidad:    Computación
  Curso:           Tercero
  Cuatrimestre:    Segundo

  Departamento:    Informática y Análisis Numérico
  Centro:          Escuela Politécnica Superior de Córdoba
  Universidad de Córdoba

  Curso acad3mico: 2020 - 2021

Fichero de ejemplo para el intérprete de pseudocódigo en español:
ipe.exe
>>

#borrar;

#lugar(3,10);
```

```

escribir_cadena('Ejemplo de cambio del tipo de valo \n');

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

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

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

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

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

```

Menu.e

```

<<
  Asignatura:      Procesadores de Lenguajes

  Titulación:      Ingeniería Informática
  Especialidad:    Computación
  Curso:           Tercero
  Cuatrimestre:    Segundo

  Departamento:    Informática y Análisis Numérico
  Centro:          Escuela Politécnica Superior de Córdoba
  Universidad de Córdoba

  Curso académico: 2020 - 2021

  Fichero de ejemplo para el intérprete de pseudocódigo en español:
ipe.exe
>>

@ Bienvenida

#borrar;

#lugar(10,10);

```

```

escribir_cadena('Introduce tu nombre --> ');

leer_cadena(nombre);

#borrar;
#lugar(10,10);

escribir_cadena(' Bienvenido/a << ');

escribir_cadena(nombre);

escribir_cadena(' >> al intérprete de pseudocódigo en
español:\'ipe.exe\'.');

#lugar(40,10);
escribir_cadena('Pulsa una tecla para continuar');
leer_cadena( pausa);

repetir

    @ Opciones disponibles

    #borrar;

    #lugar(10,10);
    escribir_cadena(' Factorial de un número --> 1 ');

    #lugar(11,10);
    escribir_cadena(' Máximo común divisor ----> 2 ');

    #lugar(12,10);
    escribir_cadena(' Finalizar -----> 0 ');

    #lugar(15,10);
    escribir_cadena(' Elige una opcion ');

    leer(opcion);

    #borrar;

    @ Fin del programa
    si (opcion = 0)
        entonces
            #lugar(10,10);
            escribir_cadena(nombre);
            escribir_cadena(': gracias por usar el intérprete ipe.exe ');

```

```

@ Factorial de un número
si_no
    si (opcion = 1)
        entonces
            #lugar(10,10);
            escribir_cadena(' Factorial de un numero ');

            #lugar(11,10);
            escribir_cadena(' Introduce un numero entero ');
            leer(N);

            factorial := 1;

            para i desde 2 hasta N paso 1 hacer
                factorial := factorial * i;
            fin_para;

        @ Resultado
        #lugar(15,10);
        escribir_cadena(' El factorial de ');
        escribir(N);
        escribir_cadena(' es ');
        escribir(factorial);

@ Máximo común divisor
si_no
    si (opcion = 2)
        entonces
            #lugar(10,10);
            escribir_cadena(' Máximo común divisor de dos
números ');

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

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

            #lugar(13,10);
            escribir_cadena(' Escribe el segundo
número ');

            leer(b);

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

```



```

                                b := auxiliar;
        fin_si;

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

        @ Se aplica el método de Euclides
        resto := a #mod b;

        mientras (resto <> 0) hacer
            a := b;
            b := resto;
            resto := a #mod b;
        fin_mientras;

        @ Se muestra el resultado
        #lugar(15,10);
        escribir_cadena(' Máximo común divisor de
');

        escribir(A1);
        escribir_cadena(' y ');
        escribir(B1);
        escribir_cadena(' es ---> ');
        escribir(b);

        @ Resto de opciones
        si_no
            #lugar(15,10);
            escribir_cadena(' Opcion incorrecta ');

        fin_si;

    fin_si;

fin_si;

#lugar(40,10);
escribir_cadena('\n Pulse una tecla para continuar --> ');
leer_cadena(pausa);

hasta (opcion = 0);

@ Despedida final

#borrar;
#lugar(10,10);
escribir_cadena('El programa ha concluido');

```

Ejemplo 1

Hemos hecho un primer ejemplo que consiste en introducir un número por pantalla y devuelve la posición correspondiente a la serie de Fibonacci.

```
@Pseudocódigo de serie de fibonacci

#borrar;
escribir_cadena('Ingrese un numero: ');
leer(dato);

a:= 1;
b:= 1;
i:=1;

Mientras (i<dato) hacer
    escribir(a);
    c:= a+b;
    a:=b;
    b:=c;
    i:=i+1;
fin_mIENTras;

escribir_cadena('El resultado es: ');
escribir(a);
```

Ejemplo 2

Este ejemplo consiste en un menú implementado con switch. Las dos opciones son dos minijuegos. El primero consiste en adivinar un número aleatorio. Como no tenemos implementada la generación de números aleatorios hemos puesto que el número a adivinar sea el 4. El segundo minijuego es un ejercicio de cálculo mental en el que se nos pondrán 5 operaciones a realizar. Por cada operación acertada obtendremos 2 puntos y por cada fallo perderemos 1 punto.

```
#borrar;

escribir_cadena('Bienvenido\n');
opcion:= 10;
```

```

mientras (opcion <> 0) hacer

    escribir_cadena('1. Adivina el numero.\n');
    escribir_cadena('2. Calculo mental.\n');
    escribir_cadena('0. Salir.\n');

    escribir_cadena('A que desea jugar --> ');
    leer(opcion);

    casos(opcion)

    valor 0:
        escribir_cadena('¡Hasta luego!\n');
    valor 1:
        #borrar;
        escribir_cadena('Tienes que adivinar el numero del 1 al 15 que voy a
pensar...\n');
        escribir_cadena('Escribe un numero --> ');
        leer(num);

        mientras (num <> 4)
            hacer
                escribir_cadena('Incorrecto\n');
                si (num > 4) entonces
                    escribir_cadena('El numero es inferior\n');
                si_no
                    escribir_cadena('El numero es superior\n');
                fin_si;
                leer(num);

        fin_mientras;
        escribir_cadena('¡Correcto!\n');

    valor 2:
        #borrar;
        escribir_cadena('Se pondrán una serie de operaciones y tendrás que
acertarlas\n');
        escribir_cadena('Cada acierto sumara 2 puntos y cada fallo restará
uno\n');
        puntos:=0;

        escribir_cadena('11*11 = ');
        leer(res);

        si (res = 121) entonces
            puntos := puntos +2;

        si_no
            puntos := puntos -1;
        fin_si;

        escribir_cadena('125 - 15 = ');
        leer(res);

        si (res = 110) entonces
            puntos := puntos +2;

        si_no
            puntos := puntos -1;
        fin_si;

```

```

        escribir_cadena('12 * 3= ');
        leer(res);

        si (res = 36) entonces
            puntos := puntos +2;

        si_no
            puntos := puntos -1;
        fin_si;

        escribir_cadena('60/5 = ');
        leer(res);

        si (res = 12) entonces
            puntos := puntos +2;

        si_no
            puntos := puntos -1;
        fin_si;

        escribir_cadena('46+64 = ');
        leer(res);

        si (res = 110) entonces
            puntos := puntos +2;

        si_no
            puntos := puntos -1;
        fin_si;

        escribir_cadena('Has conseguido un total de ');
        escribir(puntos);
        escribir_cadena('puntos\n');

    defecto:
        escribir_cadena('Escriba otro numero\n');
    fin_casos;
fin_mientras;

```

Ejemplo 3

En este ejemplo se pide por pantalla al usuario un número y se muestra la tabla de multiplicar del mismo. Este ejemplo es para ver el bucle for en un ejemplo un poco más claro.

```

#borrar;

escribir_cadena('Que tabla de multiplicar desea ver --> ');
leer(num);

si (num>=1 #y num<=10)
    entonces

```

```
    para x desde 0 hasta 10 hacer
        escribir_cadena('\n');
        escribir(x*num);

    fin_para;
si_no
    escribir_cadena('El numero debe estar entre 0 y 10\n');
fin_si;
```

10 Conclusiones

Los puntos fuertes creo que son el control de errores que hemos hecho, aunque quizá no sea muy eficiente, creo que ayuda bastante. Un detalle que me parecía correcto comentar es que al ser un lenguaje en pseudocódigo tan sencillo y traducido al castellano, para un usuario novato en la programación le podría ser de gran ayuda para empezar.

El punto más flojo sería la falta de sentencias con respecto a un lenguaje real, aunque estén las más básicas faltan muchas otras y quizás falten herramientas en caso de utilizarlo para un caso real. Aunque esto es normal debido a la falta de tiempo.

La creación de este intérprete nos ha ayudado a comprender el funcionamiento de los lenguajes de programación. Con este intérprete se comprende a nivel práctico lo que se da en esta asignatura en la parte de teoría, desde el análisis léxico hasta el semántico pasando por el léxico.

Es una experiencia totalmente nueva en cuanto a trabajos realizados durante la carrera puesto que ninguna asignatura trata este tema. Ha sido frustrante y gratificante a partes iguales. La dificultad del trabajo ha sido elevada debido a la nula experiencia que teníamos con este tipo de lenguajes como flex o yacc y la poca información que hay en internet. Aún así creo que hemos hecho un intérprete competente a nuestro nivel.

11 Bibliografía

http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf

<http://www.uco.es/users/ma1fegan/2019-2020/pl-grado/practicas/Guion-practicas.pdf>