IPE

Intérprete de Pseudocódigo en español

Procesadores de lenguajes

3º INGENIERIA INFORMÁTICA , Especialidad en Computación
2º Cuatrimestre del curso 2018-2019
Córdoba 03/06/2019
Escuela Politécnica superior de Córdoba
Universidad de Córdoba

Antonio Ariza Garcia y Enrique Galán Galán
I62argaa – i62gagae

Contenido

1. INTRODUCCIÓN

Breve descripción del trabajo realizado y de las partes del documento.

Utilizando las herramientas de Flex y Bison se ha realizado un intérprete de pseudocódigo en español “Ipe.exe”.

En el presente documento, se van a abordar los siguientes temas:

- Lenguaje de pseudocódigo: Se expondrán las distintas sentencias del lenguaje (condicionales, bucles…), las cuales estarán en castellano.

- Tabla de símbolos: Se expondrán las clases utilizadas.

- Análisis léxico: Se describirán los componentes léxicos y sus expresiones regulares.

- Análisis sintáctico: Se describirá la gramática de contexto libre del intérprete.

- Código de AST: Se detallará toda la lógica que ejecutan las distintas reglas del intérprete.

- Funciones auxiliares: Se detallarán las distintas funciones auxiliares que hayan sido necesarias en la codificación del intérprete.

- Modo de obtención del intérprete: Aquí se hablará de cómo generar el ejecutable del intérprete.

- Modo de ejecución del intérprete: Se expondrán los 2 modos de ejecución que soporta el intérprete: interactivo y a partir de un fichero.

- Ejemplos: Se presentará un conjunto de ejemplos con los cuales se podrá probar el intérprete.

2. LENGUAJE DE PSEUDOCÓDIGO

2.1 Componentes Léxicos o token

- Palabras Reservadas
    - _mod, _div
    - _o, _y, _no
    - leer, leer_cadena
    - escribir, escribir_cadena
    - si, entonces, si_no, fin_si
    - mientras, hacer, fin_mientras
    - repetir, hasta, para, fin_para, desde, paso
    - _borrar, _lugar
- Identificadores
    - Estarán compuestos por una serie de letras, dígitos y subrayado
    - Deben comenzar por una letra

- No podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.
- Identificadores válidos: dato, dato_1_a
- Identificadores no válidos: _dato, dato_, dato__1

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

- Cadena
  - Estará compuesta por una serie de caracteres delimitados por comillas simples: 'Ejemplo de cadena', 'Ejemplo de cadena con salto de línea \n y tabulador \t'.
  - Deberá permitir la inclusión de la comilla simple utilizando la barra (\): 'Ejemplo de cadena con \' comillas \' simples.
  - Las comillas exteriores no se almacenarán como parte de la cadena.

- Operador de asignación
  - Asignación: :=

- Operadores aritméticos
  - Suma: +
    - ✓ Unario: +2
    - ✓ Binario: 2+3
  - Resta: -
    - ✓ Unario: -2
    - ✓ Binario: 2-3
  - Producto: *
  - División: /
  - División entera: _div
  - Módulo: _mod
  - Potencia: **

- Operador alfanumérico
  - Concatenación: ||

- Operadores relaciones de números y cadenas
  - Menor que: <
  - Menor o iguales que: <=
  - Mayor que: >
  - Mayor o igual: >=
  - Igual que: =
  - Distinto que: <>

- Ejemplo: si A es una variable numérica y control una variable alfanumérica, se pueden generar las siguientes expresiones relacionales: (A >= 0)
  (control <> 'stop')

- Operadores Lógicos
  - Disyunción lógica: _o
  - Conjunción lógica: _y
  - Negación lógica: _no
  - Ejemplo: (A >= 0) _y _no (control <> 'stop')

- Comentarios
  - De varias líneas: delimitados por el símbolos #
    # ejemplo de
    Comentario de
    tres líneas #
  - De una línea: Todo lo que siga al carácter @ hasta el final de la línea.
    @ ejemplo de 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_cadena (*identificador*)
    - ✓ Declara a identificador como variable alfanumérica y le asigna la cadena leída (sin comillas).

- Escritura
  - Escribir (*expresión numérica)*

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

- 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 ('\t Introduzca el dato \n');
```

Sentencias de control

- Sentencia condicional simple

```
Si condición
        entonces sentencias
fin_si
```

- Sentencia condicional compuesta

```
si condición
        entonces sentencias
        si_no sentencias
fin_si
```

- Bucle "mientras"

```
mientras condición hacer
        sentencias
fin_mientras
```

- Bucle "repetir"

```
repetir
        sentencias
hasta  condición
```

- Bucle "para"

```
para identificador
        desde expresión numérica 1
        hasta expresión numérica 2
        [paso expresión numérica 3]
        Hacer
                Sentencias
fin_para
```

Comandos Especiales

- _borrar
  - ✓ Borra la pantalla
- _lugar (*expresión numérica 1, expresión numérica 2)*

✓ *Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numérica.*

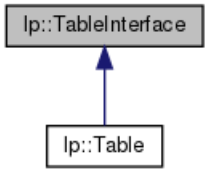
3. TABLA DE SÍMBOLOS

Las misiones principales de la Tabla de Símbolos en el proceso de traducción son:

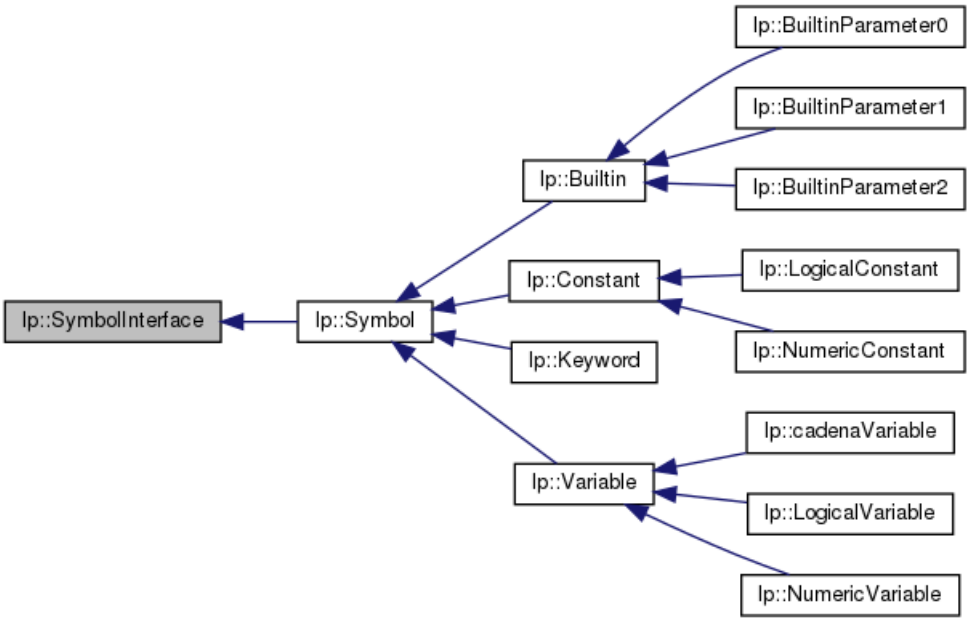
- Colaborar con las comprobaciones semánticas.
- Facilitar ayuda a la generación de código.

Resumen de las clases utilizadas

lp::Table



- Atributo privado: - std::map< std::string, lp::Symbol * > _table
- Observadores: - bool lookupSymbol (const std::string &name) const
  - lp::Symbol * getSymbol (const std::string &name)
  - bool isEmpty () const
  - int getNumberOfSymbols () const
- Modificadores: - void installSymbol (Symbol *s)
  - void eraseSymbol (const std::string &name)



lp::Symbol

- Atributo protegido: - std::string _name;
  - int _token;

- Constructores: - Symbol (std::string name="", int token=0)
  - Symbol (const Symbol &s)
- Observadores: - const std::string & getName () const
  - int getToken () const
- Modificadores : - void setName (const std::string &name)
  - void setToken (int token)
- Operadores Relacional: - bool operator== (const Symbol &s) const
  - bool operator< (const Symbol &s) const

lp::Builtin

- Atributo protegido: - int _nParameters
- Constructores : - Builtin (std::string name="", int token=0, int nParameters=0)
  - Builtin (const Builtin &b)
- Observadores: - int getNParameters () const
- Modificadores : - void setNParameters (int nParameters)
- Operadores: - virtual Builtin & operator= (const Builtin &b)
- I/O funciones: - virtual void write () const
  - virtual void read ()

lp::Constant

- Atributo protegido: - int _type
- Constructores : - Constant (std::string name="", int token=0, int type=0)
  - Constant (const Constant &c)
- Observadores: - int getType () const
- Modificadores : - void setType (int type)
- Operadores: - virtual Constant & operator= (const Constant &c)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Symbol y que se ha usado para la declaración de constantes.

lp::Keyword

- Atributo protegido: - int _type
- Constructores : - Keyword (std::string name="", int token=0)
  - Keyword (const Keyword &k)
- Operadores: - Keyword & operator= (const Keyword &k)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Symbol y que se ha usado para la declaración de palabras reservadas.

lp::Variable

- Atributo protegido: - int _type
- Constructores : - Constant (std::string name="", int token=0, int type=0)
  - Constant (const Constant &c)
- Observadores: - int getType () const
- Modificadores : - void setType (int type)
- Operadores: - virtual Constant & operator= (const Constant &c)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Symbol y ha usado para a declaración de variables.

lp::BuiltinParameter0

- Atributo privado: - lp::TypePointerDoubleFunction_0 _function
- Constructores : - BuiltinParameter0 (std::string name, int token, int nParameters, lp::TypePointerDoubleFunction_0 function)
  - BuiltinParameter0 (const BuiltinParameter0 &f)
- Observadores: - lp::TypePointerDoubleFunction_0 getFunction () const
- Modificadores : - void setFunction (const lp::TypePointerDoubleFunction_0 &function)
- Operadores: - BuiltinParameter0 & operator= (const BuiltinParameter0 &f)

Clases utilizadas para la construcción de los parámetros de la tabla.

lp::BuiltinParameter1

- Atributo privado: - lp::TypePointerDoubleFunction_1 _function
- Constructores : - BuiltinParameter1 (std::string name, int token, int nParameters, lp::TypePointerDoubleFunction_1 function)
  - BuiltinParameter1 (const BuiltinParameter1 &f)
- Observadores: - lp::TypePointerDoubleFunction_1 getFunction () const
- Modificadores : - void setFunction (const lp::TypePointerDoubleFunction_1 &function)
- Operadores: - BuiltinParameter1 & operator= (const BuiltinParameter1 &f)

lp::BuiltinParameter2

- Atributo privado: - lp::TypePointerDoubleFunction_1 _function
- Constructores : - BuiltinParameter2 (std::string name, int token, int nParameters, lp::TypePointerDoubleFunction_2 function)
  - BuiltinParameter2 (const BuiltinParameter2 &f)
- Observadores: - lp::TypePointerDoubleFunction_2 getFunction () const
- Modificadores : - void setFunction (const lp::TypePointerDoubleFunction_2 &function)
- Operadores: - BuiltinParameter2 & operator= (const BuiltinParameter2 &f)

lp::LogicalConstant

- Atributo privado: - bool _value
- Constructores : - LogicalConstant (std::string name="", int token=0, int type=0, bool value=true)
  - BuiltinParameter2 (const BuiltinParameter2 &f)
- Observadores: - bool getValue () const
- Modificadores : - void setValue (const bool &value)
- Operadores: - LogicalConstant & operator= (const LogicalConstant &n)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Constant y que se ha usado para la declaración de constantes booleanas.

lp::NumericConstant

- Atributo privado: - double _value
- Constructores : - NumericConstant (std::string name="", int token=0, int type=0, double value=0.0)
  - NumericConstant (const NumericConstant &n)
- Observadores: - double getValue () const
- Modificadores : - void setValue (const double &value)
- Operadores: - NumericConstant & operator= (const NumericConstant &n)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Constant y que se ha usado para la declaración de constantes numéricas.

lp::LogicalVariable

- Atributo privado: - bool _value
- Constructores : - LogicalVariable (std::string name="", int token=0, int type=0, bool value=false)
  - LogicalVariable (const LogicalVariable &n)
- Observadores: - bool getValue () const
- Modificadores : - void setValue (bool value)
- Operadores: - LogicalVariable & operator= (const LogicalVariable &n)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Constant y que se ha usado para la declaración de Variables Booleanas.

lp::NumericVariable

- Atributo privado: - double _value
- Constructores : - NumericVariable (std::string name="", int token=0, int type=0, double value=0.0)
  - NumericVariable (const NumericVariable &n)
- Observadores: - double getValue () const
- Modificadores : - void setValue (const double &value)
- Operadores: - NumericVariable & operator= (const NumericVariable &n)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Variable y que se ha usado para declarar variables numéricas.

lp::CadenaVariable

- Atributo privado: - double _value
- Constructores : - cadenaVariable (std::string name="", int token=0, int type=0, string value="")
  - cadenaVariable (const cadenaVariable &n)
- Observadores: - string getValue () const
- Modificadores : - void setValue (const string &value)
- Operadores: - cadenaVariable & operator= (const cadenaVariable &n)
- I/O funciones: - virtual void write () const
  - virtual void read ()

Clase que hereda de Variable y que se ha usado para declarar variables de cadenas.

4. ANÁLISIS LÉXICO

4.1 Definiciones regulares

- DIGITO

Expresión Regular: [0-9]

- LETTER

Expresión Regular: [a-zA-Z]

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

- Identificadores válidos: dato, dato_1_a
- Identificadores no válidos: _dato, dato_, dato__1

Expresión Regular: ([a-z]|[A-Z]|"ñ"|"Ñ")+(({DIGIT}|"_")?({DIGIT}|{LETTER}))*

- ([a-z]|[A-Z]|"ñ"|"Ñ"): Con esta expresión al principio aseguramos que un identificador no empiece por otra cosa que no sea una letra, minúscula o mayúscula, incluida la ñ. La doble comilla indica la cadena literal.
- + : una o más veces lo anterior, es decir, el paréntesis.
- ({DIGIT}|"_")? : Un digito o un guion bajo y la ? quiere decir cero o una vez lo anterior.
- ({DIGIT}|{LETTER}) : Un digito o una letra.
- (({DIGIT}|"_")?({DIGIT}|{LETTER}))* : Con los paréntesis agrupamos y * es para cero o más veces lo anterior.

Número

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

Expresión Regular: {DIGIT}+(\.{DIGIT}+)?(E[+\-]?{DIGIT}+)?

- {DIGIT}+ : Un digito una o mas veces.
- (\.{DIGIT}+)? : La \ se utiliza para coger el siguiente carácter literal, en nuestro caso un . , le sigue un digito una o mas veces porque esta el + . La ? para que este paréntesis este cero o solo una vez.
- (E[+\-]?{DIGIT}+)? : Una E mayúscula, los corchetes se usan [+\-] para una clase de caracteres , empareja con un +, una \ o un - . ? cero o solo una vece lo anterior, es este caso los corchetes. Por último un digito una o mas veces y el último ? engloba al paréntesis para que empareje cero o más veces.

Cadena

- Estará compuesta por una serie de caracteres delimitados por comillas simples: ‘Ejemplo de cadena’, ‘Ejemplo de cadena con salto de línea \n y tabulador \t’.
- Deberá permitir la inclusión de la comilla simple utilizando la barra (\): ‘Ejemplo de cadena con \’ comillas \’ simples.
- Las comillas exteriores no se almacenarán como parte de la cadena.

Expresión Regular: """([^']|"\\'"|"\n"|"\t")*"""

- Debe comenzar por comilla simple, el [^’] una clase de caracteres negada, es decir cualquier cosa menos la comilla, con la doble “\\’” con esto permitimos la \’ en la cadena y con lo demás el salto de línea y el tabulador, todo el paréntesis cero o mas veces * , y para finalizar una comilla simple.

- Comentario en línea
    - Todo lo que siga @ hasta el final de línea

Expresión Regular: \@.*

- Utilizamos la \ para empareje con el siguiente carácter literal, es decir, con el @
- El . lo utilizamos porque indica cualquier carácter menos una línea nueva.
- * para indicar cero o mas veces lo anterior, es decir el . que es cualquier cosa menos línea nueva.

- Comentario multilínea
    - De varias líneas: delimitados por el símbolos #

Expresión Regular: "#"(.|\n)*"#"

- Con la "#" indicamos que debe emparejar con la #
- (.|\n) Cualquier cosa menos una línea nueva, y por lo tanto ponemos también el \n, con este paréntesis indicamos que empareje con cualquier cosa.
- Con * cero o más veces lo anterior
- Para terminar debe aparecer otra vez #

4.2 Expresiones regulares

- Operador de asignación
    - :=

Expresión Regular: ":="

- Operadores aritméticos
    - Suma: +

Expresión Regular: "+"

    - Resta: -

Expresión Regular: "-"

    - Producto: *

Expresión Regular: "*"

    - Division: /

Expresión Regular: "/"

    - División entera: _div

Expresión Regular: "_div"

    - Módulo: _mod

Expresión Regular: "_mod"

- Potencia: **

Expresión Regular: "**"

- Operadores alfanumérico:
  - Concatenación: ||

Expresión Regular: "||"

- Operadores relacionales de números y cadenas:
  - Menor que: <

Expresión Regular: "<"

- Menor o igual que: <=

Expresión Regular: "<="

- Mayor que: >

Expresión Regular: ">"

- Mayor o que: >=

Expresión Regular: ">="

- Igual que: =

Expresión Regular: "="

- Distinto que: <>

Expresión Regular: "<>"

- Operadores lógicos:
  - Disyunción lógica: _o

Expresión Regular: "_o"

- Conjunción lógica: _y

Expresión Regular: "_y"

- Mayor que: _no

Expresión Regular: "_no"

- Punto y coma:
    - Fin de sentencia

Expresión Regular: ";"

5. ANÁLISIS SINTÁCTICO

Descripción de la gramática de contexto libre

- Símbolos de la gramática

✓ Símbolos terminales (componentes léxicos)

✓ Símbolos no terminales

9

- Reglas de producción de la gramática
- Acciones semánticas:

✓ Se deberán describir las acciones semánticas de las producciones que generan las sentencias de control y especialmente las diseñadas para los bucles "**repetir**" y "**para**".

✓ Se valorará la inclusión de gráficos explicativos.

o **Observación**

- Véase el fichero de YACC

- SÍMBOLOS DE LA GRAMÁTICA

o SÍMBOLOS TERMINALES

✓ PRINT
✓ READ
✓ IF
✓ ELSE
✓ WHILE
✓ DO
✓ END_WHILE
✓ READ_STRING
✓ PRINT_STRING
✓ THEN END_IF
✓ REPEAT
✓ UNTIL
✓ FOR
✓ SINCE
✓ END_FOR
✓ STEP
✓ BORRAR
✓ LUGAR
✓ LETFCURLYBRACKET
✓ RIGHTCURLYBRACKET
✓ SEMICOLON

- ✓ VARIABLE
- ✓ UNDEFINED
- ✓ CONSTANT
- ✓ BUILTIN
- ✓ BOOL
- ✓ NUMBER
- ✓ CADENA
- ✓ COMMA
- ✓ ASSIGNMENT
- ✓ OR
- ✓ AND
- ✓ GREATER_OR_EQUAL
- ✓ LESS_OR_EQUAL
- ✓ GREATER_THAN
- ✓ LESS_THAN
- ✓ EQUAL
- ✓ NOT_EQUAL
- ✓ NOT
- ✓ PLUS
- ✓ MINUS
- ✓ MULTIPLICATION
- ✓ DIVISION
- ✓ MODULO
- ✓ DIVISION_ENTERA
- ✓ CONCAT
- ✓ LPAREN
- ✓ RPAREN
- ✓ POWER
- ✓ UNARY

- SIMBOLOS NO TERMINALES
  - ✓ stmt
  - ✓ asgn
  - ✓ print
  - ✓ read
  - ✓ if
  - ✓ while
  - ✓ read_string
  - ✓ print_string
  - ✓ repeat
  - ✓ for
  - ✓ stmtlist
  - ✓ listOfExp
  - ✓ restOfListOfExp
  - ✓ exp
  - ✓ cond

- Reglas de producción de la gramática

Una regla general en Bison tiene la siguiente forma general

Result: componente…

Donde Result representa un nombre no terminal, y componente son símbolos terminales y no terminales que se combinan con esta regla. Por ejemplo,

exp: exp '+' exp;

dice que dos agrupaciones de tipo exp, con un token "+" en medio, se pueden combinar en un

mayor agrupación de tipo exp. Reglas de IPE. Una acción se ve así:

{Declaraciones de C}

- ✓ program : stmtlist
  - ➢ Crea una nueva AST y la asigna a la raíz
- ✓ stmtlist:
  - ➢ Crea una lista vacía de sentencias
- ✓ stmtlist: stmtlist stmt
  - ➢ Añade una sentencia a la lista sentencias
- ✓ stmtlist : stmtlist error
  - ➢ Copia la lista cuando se produce un error
- ✓ stmt: SEMICOLON
  - ➢ Crea una sentencia vacía
- ✓ stmt: asgn SEMICOLON
  - ➢ Una Sentencia puede ser asignación y simplificamos la regla de asignación factorizando el punto y coma.
- ✓ stmt: print SEMICOLON
  - ➢ Una Sentencia puede ser una escritura y simplificamos la regla de asignación factorizando el punto y coma.
- ✓ stmt: read SEMICOLON
  - ➢ Una Sentencia puede ser una lectura y simplificamos la regla de asignación factorizando el punto y coma.
- ✓ stmt: if
  - ➢ Decimos que una sentencia puede ser un condicional.
- ✓ stmt: while
  - ➢ Decimos que una sentencia puede ser un bucle mientras

- ✓ stmt: read_string SEMICOLON
  - ➢ Una sentencia puede ser una función de lectura de cadena y la simplificamos factorizando el punto y coma.

- ✓ stmt: print_string SEMICOLON
  - ➢ Una sentencia puede ser una función de escritura de cadena y la simplificamos factorizando el punto y coma.

- ✓ stmt: repeat
  - ➢ Decimos que una sentencia puede ser un bucle repeat.

- ✓ stmt: for
  - ➢ Decimos que una sentencia puede ser un bucle para.

- ✓ stmt: BORRAR SEMICOLON
  - ➢ Una sentencia puede ser una función de borrar y la simplificamos factorizando el punto y coma.

- ✓ stmt: LUGAR LPAREN exp COMMA exp RPAREN SEMICOLON
  - ➢ Una sentencia puede ser una función de lugar y la simplificamos factorizando el punto y coma.

- ✓ if: IF cond THEN stmtlist END_IF
  - ➢ Un condicional está compuesto por un token IF, una condición, un token THEN, una lista de sentencias y un token END_IF.

- ✓ if: IF cond THEN stmtlist ELSE stmtlist END_IF
  - ➢ Un condicional está compuesto por un token IF, una condición, un token THEN, una lista de sentencias un token ELSE, otra lista de sentencias y un token END_IF.

- ✓ repeat: REPEAT stmtlist UNTIL cond
  - ➢ Token REPEAT, una lista de sentencias, un token UNTIL y una condición

- ✓ while: WHILE cond DO stmtlist END_WHILE
  - ➢ Token WHILE, una condición, token DO, una lista de sentencias y un token END_WHILE

- ✓ for: FOR VARIABLE SINCE exp UNTIL exp DO stmtlist END_FOR
  - ➢ Un token FOR, VARIABLE, SINCE, una expresión, un token DO, una lista sentencias y un token END_FOR

- ✓ cond: LPAREN exp RPAREN
  - ➢ Una condición estará formada por un token LPAREN, una expresión, un token RPAREN

- ✓ asgn: VARIABLE ASSIGNMENT exp
  - ➢ Token VARIABLE, ASSIGMENT y una expresión.

- ✓ asgn : VARIABLE ASSIGNMENT asgn
  - ➢ Token VARIABLE, ASSIGMENT y una asgn.

- ✓ asgn: CONSTANT ASSIGNMENT exp
  - ➢ Token CONSTANT ASSIGNMENT y una expresión

- ✓ asgn: CONSTANT ASSIGNMENT asgn
  - ➢ Token CONSTANT ASSIGNMENT y una asgn

- ✓ print: PRINT exp
  - ➢ Token PRINT y una expresión.

- ✓ print_string: PRINT_STRING exp
  - ➢ Un token PRINT_STRING y una expresión

- ✓ read: READ LPAREN VARIABLE RPAREN
  - ➢ Regla que crea un sentencia de lectura a partir Token READ, LPAREN, VARIABLE y RPAREN.

- ✓ read: READ LPAREN CONSTANT RPARE
  - ➢ Regla que produce un error al intentar leer una constante.

- ✓ read_string: READ_STRING LPAREN VARIABLE RPAREN
  - ➢ Regla que crea un sentencia de escritura a partir Token READ_STRING, LPAREN, VARIABLE y RPAREN.

- ✓ read_string: READ_STRING LPAREN CONSTANT RPAREN
  - ➢ Regla que produce un error al intentar leer una cadena constante.

6. CÓDIGO AST

Resumen de las clases utilizadas

Este código se divide en dos ficheros, un fichero ".hpp" que nos proporciona las declaraciones de las clases y el prototipo de sus funciones y por otro lado el fichero ".cpp" en el que encontramos la codificación de las funciones de nuestras clases, debemos destacar que todas las clases poseen una función print(), cuya función es imprimir nuestra expresión leída del código y una función evaluate() la cual se encarga de evaluar esa expresión leída y darle significado según lo codificado.

Las clases que tenemos son:

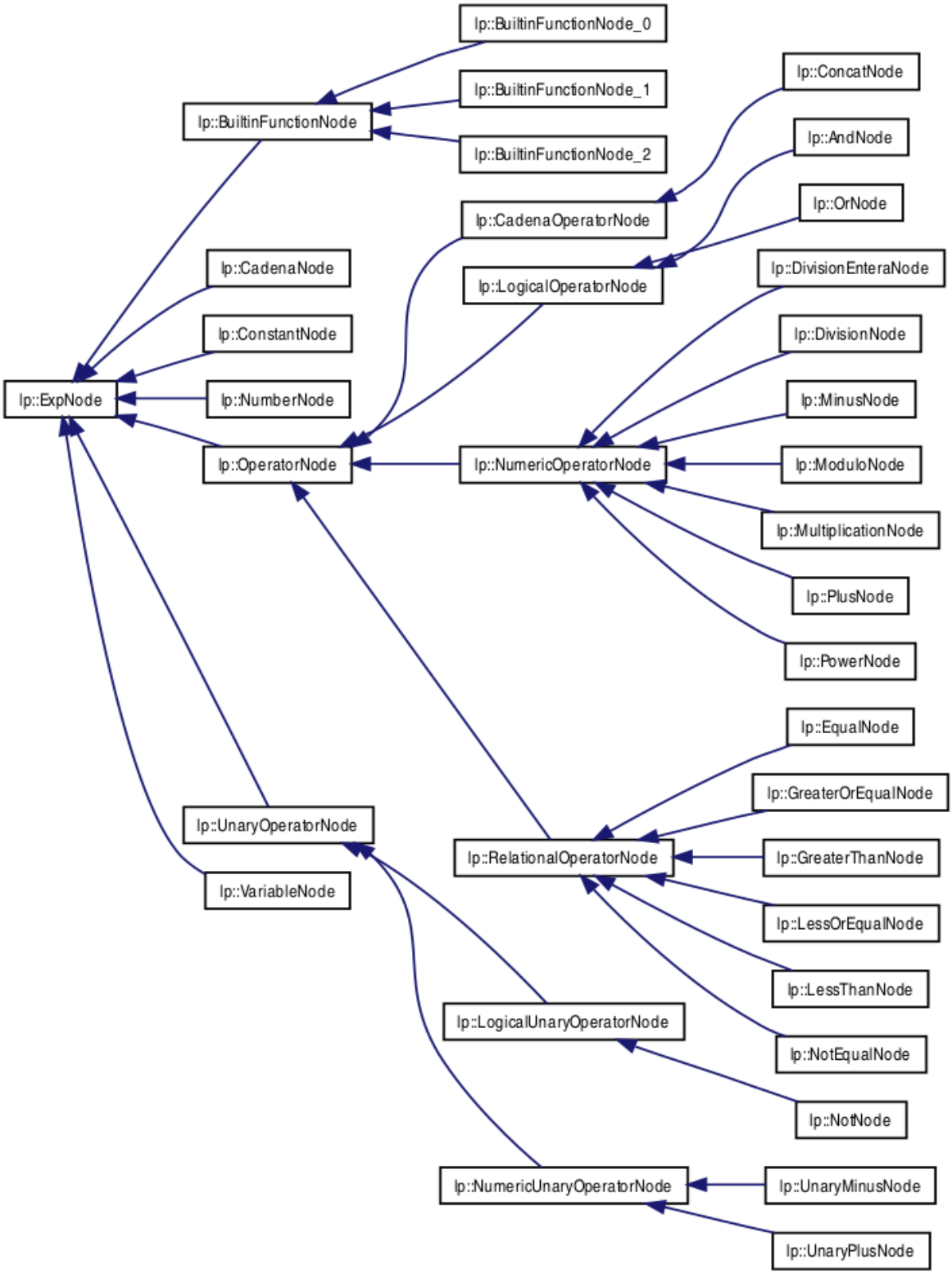
- ✓ ExpNode: Clase que nos representa una expresión de nuestro lenguaje, esta clase posee cinco funciones virtuales en las cuales se evalúan los distintos tipos de expresiones leídas.

    - ✓ VariableNode: Clase que hereda de ExpNode, esta clase representa una variable de nuestro código, por lo tanto, su atributo es el ID de esta.
    - ✓ ConstantNode: Clase que hereda de ExpNode, esta clase nos representa una constante de nuestro código ya se de tipo numérica o lógica.
    - ✓ NumberNode: Clase que hereda de ExpNode, esta clase nos representa todo número leído de nuestro código, sea real, entero o escrito en notación científica.
    - ✓ CadenaNode: Clase que hereda de ExpNode, esta clase lo que nos representa son las cadenas alfanuméricas leídas en nuestro código.
    - ✓ -UnaryOperatorNode: Clase que hereda de ExpNode, esta clase nos representa los operadores unarios que ya veremos más adelante.
    - ✓ NumericUnaryOperatorNode: Clase que hereda de UnaryOperatorNode, esta clase represente los operadores numéricos unarios.
    - ✓ -LogicalUnaryOperatorNode: Clase que hereda de UnaryOperatorNode, esta clase represente los operadores lógicos unarios.
    - ✓ -UnaryMinusNode: Clase que hereda de NumericUnaryOperatorNode, esta clase representa el operador de resta numérico.
    - ✓ -UnaryPlusNode: Clase que hereda de NumericUnaryOperatorNode, esta clase representa el operador de suma numérico.
    - ✓ OperatorNode: Clase que hereda de ExpNode, esta clase representa cualquier operador de nuestro código ya sea numérico, alfanumérico o lógico.
    - ✓ NumericOperatorNode: Clase que hereda de OperatorNode, esta clase representa aquellos operadores de tipo exclusivo numérico.
    - ✓ CadenaOperatorNode: Clase que hereda de OperatorNode, esta clase representa un operador de tipo alfanumérico tal como la concatenación.
    - ✓ RelationaOperatorNode: Clase que hereda de OperatorNode, esta clase representa los operadores relacionales de nuestro código.
    - ✓ LogicalOperatorNode: Clase que hereda de OperatorNode, esta clase representa los operadores lógicos de nuestro código.
    - ✓ PlusNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de suma numérico.
    - ✓ MinusNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de resta numérico.
    - ✓ MultiplicationNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de producto numérico.
    - ✓ DivisionNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de división numérico.

- ✓ DivisionEnteraNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de división entera numérico el cual nos devuelve la parte entera de una división.
- ✓ ModuloNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de módulo numérico, devolviéndonos el resto de una división.
- ✓ PowerNode: Clase que hereda de NumericOperatorNode, esta clase representa el operador de potencia numérico.
- ✓ ConcatNode: Clase que hereda de CadenaOperatorNode, esta clase representa el operador de concatenación alfanumérico.
- ✓ BuiltinFunctionNode: Clase que hereda de ExpNode, representa un parámetro en una función.
- ✓ GreatherThanNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de mayor que, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ GreatherThanNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de mayor que, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ GreatherOrEqualNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de mayor o igual, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ LessThanNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de menor que, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ LessOrEqualNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de menos o igual, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ EqualNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de igualdad, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ NotEqualNode: Clase que hereda de RelationalOperatorNode, esta clase representa el operador relacionas de no igualdad, este operador nos compara tanto valores numéricos como alfanuméricos.
- ✓ AndNode: Clase que hereda de LogicalOperatorNode, esta clase representa el operador lógico AND que devuelve TRUE O FALSE en función de las expresiones analizadas.
- ✓ OrNode: Clase que hereda de LogicalOperatorNode, esta clase representa el operador lógico Or que devuelve TRUE O FALSE en función de las expresiones analizadas.
- ✓ NotNode: Clase que hereda de LogicalOperatorNode, esta clase representa el operador lógico NOT que niega la condición de la expresión.
- ✓ Statement: Clase que representa una sentencia de nuestro código, ya sea un bucle o una asignación.
- ✓ AssignmentStmt: Clase que hereda de Statement, esta clase representa una asignación en la cual a una variable se le asigna un valor.

- ✓ PrintStmt: Clase que hereda de Statement, esta clase su función principal es imprimirnos por pantalla un valor de una expresión numérica.
- ✓ ReadStmt: Clase que hereda de Statement, esta clase su función principal es leer por teclado el valor de una expresión numérica y asignárselo a una variable.
- ✓ Print_StringStmt: Clase que hereda de Statement, esta clase su función principal es imprimirnos por pantalla un valor de una expresión alfanumérica.
- ✓ Read_StringStmt: Clase que hereda de Statement, esta clase su función principal es leer por teclado el valor de una expresión alfanumérica y asignárselo a una variable.
- ✓ EmptyStmt: Clase que hereda de Statement, esta clase representa una sentencia vacía en nuestro código.
- ✓ IfStmt: Clase que hereda de Statement, esta clase nos representa un bucle IF en nuestro código, tiene dos constructores ya que puede poseer o no la opción alternativa.
- ✓ WhileStmt: Clase que hereda de Statement, esta clase nos representa un bucle While en nuestro código.
- ✓ RepeatStmt: Clase que hereda de Statement, esta clase nos representa un bucle que hace repetir una sentencia hasta que se satisfaga una determinada condición, ejecutando la sentencia siempre al menos una vez.
- ✓ ForStmt: Clase que hereda de Statement, esta clase nos representa un bucle FOR en nuestro código, esta clase tiene dos constructores ya que el paso se puede asignar o no, en caso de que no, este tomará un valor por defecto.
- ✓ BorrarStmt: Clase que hereda de Statement, esta clase cumple la función de limpiar la pantalla.
- ✓ LugarStmt: Clase que hereda de Statement, esta clase cumple la función de colocarnos el cursor en una determinada posición en la pantalla.

Grafo que nos muestra las herencias de nuestro fichero AST.



7. FUNCIONES AUXILIARES

Se han codificado una serie de funciones auxiliares matemáticas que se encuentran declaradas en el fichero mathFunction.hpp y codificadas en el fichero mathFunction.cpp entre ellas encontramos:

-Log (double x): Calcula y devuelve el logaritmo neperiano de x.
-Log10 (double x): Calcula y devuelve el logaritmo decimal de x.
-Exp (double x): Calcula el exponencial de un número real.
-Sqrt (double x): Calcula la raíz cuadrada de un número real.
-Integer(double x): Devuelve la parte entera de un número real.
-Random(): Nos devuelve un número aleatorio.
-Atan2(double x, double y): Nos devuelve el arco tangente de dos números reales.

Por otro lado, también hemos codificado una serie de funciones alfanuméricas que nos permite el tratado de cadenas, en concreto son:

Std:string tranforma(std::string x): Función que nos transforma el string recibido a minúscula para así ingresarlo o buscarlo en nuestra tabla de símbolos.

8. MODO DE OBTENCIÓN DEL INTÉRPRETE

Nuestro interprete va a estar compuesto principalmente por cinco directorios en la cual se encuentran varios ficheros:

-Directorio table: En este fichero podemos encontrar la declaración, los atributos y las funciones de cada clase que hemos utilizado en nuestro intérprete, además encontramos el fichero init.hpp donde se declaran todas las constantes y palabras reservadas.

-Builtin.hpp: Definición de los atributos y métodos de la clase builtin.
-Builtin.cpp: Código de algunas funciones de la clase builtin.

-builtinParameter0.hpp: Definición de los atributos y métodos de la clase builtinParameter0.
-builtinParameter0.cpp: Código de algunas funciones de la clase builtinParameter0.

-builtinParameter1.hpp: Definición de los atributos y métodos de la clase builtinParameter0.
-builtinParameter1.cpp: Código de algunas funciones de la clase builtinParameter0.

-builtinParameter2.hpp: Definición de los atributos y métodos de la clase builtinParameter0.
-builtinParameter2.cpp: Código de algunas funciones de la clase builtinParameter0.

-cadenaVariable.hpp: Definición de los atributos y métodos de la clase cadenaVariable, esta clase se utilizará para el tratado de cadenas en el intérprete.
-cadenaVariable.cpp: Se encuentra el código de algunas funciones de la clase cadenaVariable.

-constant.hpp: Definición de los atributos y métodos de la clase constant, esta clase se utilizará para el tratado de constantes en el intérprete.
-constant.cpp: Se encuentra el código de algunas funciones de la clase constant.

-keyword.hpp: Definición de los atributos y métodos de la clase keyword, esta clase se utilizará para el tratado de palabras clave en el intérprete.
-keyword.cpp: Se encuentra el código de algunas funciones de la clase keyword.

-logicalConstant.hpp: Definición de los atributos y métodos de la clase logicalConstant, esta clase se utilizará para el tratado de constantes lógicas en el intérprete.
-logicalConstant.cpp: Se encuentra el código de algunas funciones de la clase logicalConstant.

-logicalVariable.hpp: Definición de los atributos y métodos de la clase logicalVariable, esta clase se utilizará para el tratado de variables de tipo lógicas en el intérprete.
-logicalVariable.cpp: Se encuentra el código de algunas funciones de la clase logicalVariable.

-mathFunction.hpp: Se encuentran los prototipos de algunas funciones matemáticas.
-mathFunction.cpp: Se encuentra el código de las funciones del fichero mathFunction.hpp.

-alphaFunction.hpp: Se encuentran los prototipos de algunas funciones alfanuméricas.
-alphaFunction.cpp: Se encuentra el código de las funciones del fichero alphaFunction.hpp.

-numericConstant.hpp: Definición de los atributos y métodos de la clase numericConstant, esta clase se utilizará para el tratado de constantes numéricas en el intérprete.
-numericConstant.cpp: Se encuentra el código de algunas funciones de la clase numericConstant.

-numericVariable.hpp: Definición de los atributos y métodos de la clase numericVariable, esta clase se utilizará para el tratado de variables numéricas en el intérprete.
-numericVariable.cpp: Se encuentra el código de algunas funciones de la clase numericVariable.

-symbol.hpp: Definición de los atributos y métodos de la clase symbol, esta clase se utilizará para el tratado de símbolos en el intérprete.
-symbol.cpp: Se encuentra el código de algunas funciones de la clase symbol.

-table.hpp: Definición de los atributos y métodos de la clase table, esta clase se utilizará para la construcción de la tabla de símbolos en el intérprete.
-table.cpp: Se encuentra el código de algunas funciones de la clase table.

-variable.hpp: Definición de los atributos y métodos de la clase variable, esta clase se utilizará para el tratado de variables en el intérprete.
-variable.cpp: Se encuentra el código de algunas funciones de la clase variable.

-init.hpp: Definición de todas las constantes y palabras clave del intérprete.
-init.cpp: Se encuentra el código que inicializa todo lo declarado en init.hpp

-Directorio parser: Contiene dos ficheros:

-Interpreter.l: Fichero de escáner léxico donde se encuentran declaradas las expresiones regulares y sus correspondientes acciones.

-Interpreter.y: Fichero gramático donde se realizan las reglas de nuestro análisis léxico.

Al compilar estos dos ficheros generaremos los ficheros interpreter.tab.c e interpreter.tab.h que son los correspondientes a la tabla de símbolos.

-Directorio includes: Contiene un único fichero:

-macros.hpp: Contiene la declaración de macros utilizadas en el intérprete, tales como CLEAR_SCREEN o PLACE(x,y).

-Directorio error: Contiene dos ficheros:
-error.hpp: Contiene el prototipo de las funciones de error utilizada en el intérprete.
-error.cpp: Contiene el código de las funciones declaradas en el fichero error.hpp

-Directorio ast: Contiene dos ficheros:

-ast.hpp: Definición de todas las clases utilizadas en el intérprete, así como sus atributos y el prototipo de sus funciones.
-ast.cpp: Contiene el código de las funciones de las clases declaradas en el fichero ast.cpp

Además, también encontramos el fichero interpreter.cpp en el cual encontramos la función main de nuestro programa y en el cual se diferencia el funcionamiento del intérprete entre modo interactivo y modo por fichero.

Para compilar nuestro programa hemos utilizado varios ficheros makefile, uno global el cual va entrando a cada directorio y ejecutando el makefile propio de cada directorio de los cuales hablaremos uno a uno más adelante.
Además en el makefile podemos encontrar la opción clean la cual nos limpia todo lo generados así como los ejecutables y los ficheros ".o", también encontramos la opción "doc" que nos genera automáticamente toda la documentación que hemos ido incluyendo de doxygen en los ficheros codificados.

Entre los ficheros makefile individuales de cada directorio debemos destacar que cada uno compila los ficheros que se encuentran en el interior de cada directorio por separado, lo que nos ha facilitado la búsqueda de errores en la etapa de codificación.

9. MODO DE EJECUCIÓN DEL INTÉRPRETE

Nuestro intérprete posee dos modos de ejecución:

-Modo interactivo: Se accede escribiendo en el terminal ./interprete.exe, este modo nos permite interactuar con nuestro intérprete a nuestro antojo, programando sobre la marcha, nos permite crear variables, asignarles un valor, realizar bucles… A la hora de utilizar nuestro modo interactivo debemos de tener en cuenta que toda sentencia que debamos realizar ya sea un bucle, una asignación a una variable, imprimir por pantalla un valor numérico o alfanumérico, debemos de realizarla toda en la misma línea ya que nuestro interprete en el modo interactivo a diferencia del modo por fichero nos lee hasta el final de línea, no emparejando con las expresiones regulares que se puedan escribir en la siguiente línea. Por lo demás el modo de funcionamiento es igual al de mediante fichero.

-Modo por fichero: Este modo el cual se accede de igual manera que el anterior a diferencia de que en la línea de argumentos también debemos de incluir el fichero el cual queremos interpretar. Para que nuestro intérprete pueda tratar el fichero con corrección debemos asegurarnos como es lógico de este programado correctamente de acuerdo con nuestro lenguaje establecido, mediante el uso de este modo hemos probado nuestro ejemplo propio y los demás ejemplos del profesor como ya comentaremos en el siguiente apartado.

10. EJEMPLOS

Para probar nuestro programa y comprobar que todo funciona correctamente hemos realizado un pequeño ejemplo situado en la carpeta “\ejemplos”, este programa nos simula un menú de gestión de una biblioteca en la cual se nos muestra una serie de libros disponibles que podemos ir sacando o no en función de si están o no disponibles, se trata de un ejemplo sencillo en el cual el menú tiene dos opciones, la primera opción que nos muestra todos aquellos libros que se encuentren disponibles, es decir, que la cantidad de estos sea superior a 0. Además, tenemos otra opción en la cual podemos seleccionar el libro que deseamos sacar de la biblioteca.
Para probar además nuestro interprete hemos utilizado también todos los ejemplos utilizados como el profesor, ejemplo_1_saluda.e, ejemplo_2_factorial.e, ejemplo_3_mcd.e, ejemplo_4_menu_inicial.e, ejemplo_5_menu_completo.e, ejemplo_6_Intercambiar_tipo.e.

11. CONCLUSIONES

11.1 Reflexión sobre el trabajo realizado.

Este trabajo en su mayor parte nos ha servido para comprender el funcionamiento de un intérprete, en nuestro caso de pseudocódigo, desde su análisis léxico y reconocimiento de expresiones regulares hasta el análisis sintáctico, pasando por la generación de token y la creación de la tabla de símbolos.
Además, también nos ha servido para la continua mejora de nuestra programación en c++

11.2 Puntos fuertes y puntos débiles del intérprete desarrollado.

En cuanto a los puntos fuertes de nuestro intérprete debemos mejorar que mejora muchísimo la comprensión del código a simple vista al ser capaz de reconocer seudocódigo, de esta manera podríamos tener una programación algo más intuitiva que la actual, aunque aún quedaría por incluir muchísimas mejoras como el uso de vectores.

En cuanto a sus puntos débiles debemos destacar que aunque nos permita un modo interactivo que nosotros hemos usado sobre todo para ir probando y comprobando aquello que íbamos programando, en este modo se debe de programar todo en la misma línea, ya sea una asignación un bucle interactivo o un condicional, lo que dificulta mucho su programación.

12. BIBLIOGRAFÍA O REFERENCIAS WEB

- The Lex and Yacc Page: http://dinosaur.compilertools.net/

- Ejemplos Bison, propuestos por el profesor Nicolás Luis Fernández García: https://moodle.uco.es/m1819/pluginfile.php/1539/course/section/46907/bison.zip

- The Yacc-compatible Parser Generator
https://www.gnu.org/software/bison/manual/bison.pdf

- Temario Procesadores Lenguajes 2019 – Nicolás Luis Fernández García. Escuela Politécnica de Córdoba.

https://moodle.uco.es/m1819/course/view.php?id=1292 – Solo matriculados.