Antonio Ariza Garcia

[Dirección de correo electrónico]

Descripción breve

[Dibujar su lector con un resumen de la participación. Normalmente es un breve resumen del documento.   
Cuando esté listo para agregar contenido, haga clic aquí y empiece a escribir.]

[Título del documento]

[Subtítulo del documento]

Contenido

[1. INTRODUCCIÓN 2](#_Toc10206321)

[2. LENGUAJE DE PSEUDOCÓDIGO 2](#_Toc10206322)

[2.1 Componentes Léxicos o token 2](#_Toc10206323)

[ Palabras Reservadas 2](#_Toc10206324)

[ Identificadores 2](#_Toc10206325)

[ Número 3](#_Toc10206326)

[ Cadena 3](#_Toc10206327)

[ Operador de asignación 3](#_Toc10206328)

[ Operadores aritméticos 3](#_Toc10206329)

[ Operador alfanumérico 3](#_Toc10206330)

[ Operadores relaciones de números y cadenas 3](#_Toc10206331)

[ Operadores Lógicos 4](#_Toc10206332)

[ Comentarios 4](#_Toc10206333)

[ Punto y coma 4](#_Toc10206334)

[2.2 Sentencias 4](#_Toc10206335)

[ Asignación 4](#_Toc10206336)

[ Lectura 4](#_Toc10206337)

[ Escritura 4](#_Toc10206338)

[ Sentencias de control 5](#_Toc10206339)

[ Comandos Especiales 5](#_Toc10206340)

[3. TABLA DE SÍMBOLOS 6](#_Toc10206341)

[4. ANÁLISIS LÉXICO 10](#_Toc10206342)

[ Identificadores 10](#_Toc10206343)

[ Numero 10](#_Toc10206344)

[ Cadena 10](#_Toc10206345)

[ Comentario en línea 11](#_Toc10206346)

[ Comentario multilínea 11](#_Toc10206347)

[ Operador de asignación 11](#_Toc10206348)

[ Operadores aritméticos 11](#_Toc10206349)

[ Operadores alfanumérico: 11](#_Toc10206350)

[5. ANÁLISIS SINTÁCTICO 12](#_Toc10206351)

[6. CÓDIGO AST 12](#_Toc10206352)

[7. FUNCIONES AUXILIARES 12](#_Toc10206353)

[8. MODO DE OBTENCIÓN DEL INTÉRPRETE 12](#_Toc10206354)

[9. MODO DE EJECUCIÓN DEL INTÉRPRETE 13](#_Toc10206355)

[10. EJEMPLOS 13](#_Toc10206356)

[11. CONCLUSIONES 13](#_Toc10206357)

[12. BIBLIOGRAFÍA O REFERENCIAS WEB 13](#_Toc10206358)

[13. ANEXOS 13](#_Toc10206359)

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”.

2. LENGUAJE DE PSEUDOCÓDIGO

Se corresponde con el apartado Nº 3 de este documento

Componentes léxicos

Sentencias

o **Observaciones**

Se valorará la inclusión de ejemplos de los componentes léxicos y las sentencias

Si se ha ampliado el lenguaje de pseudocódigo entonces se deberá indicar en este apartado.

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

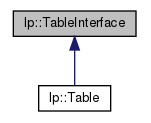
Las misiones principales de la Tabla de Símbolos en el proceso de traducción son:

• 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

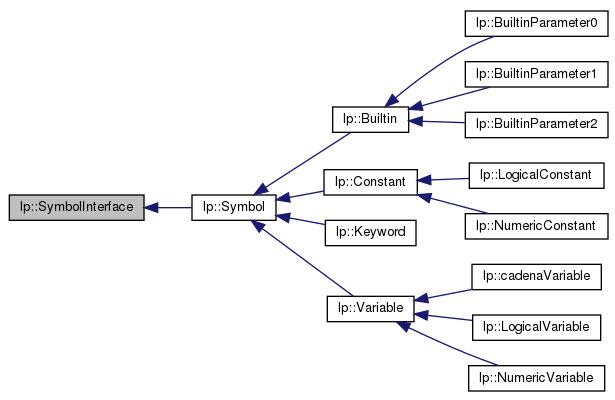
- lp::Symbol \* getSymbol (const std::string &name)

- 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 ()

**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 ()

**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 ()

**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)

**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 ()

**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 ()

**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 ()

**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 ()

**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 ()

4. ANÁLISIS LÉXICO

Descripción de los componentes léxicos y de sus expresiones regulares.

o **Observación**

Véase el fichero de flex.

* 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

**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 #
* 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:** ";"

* 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

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

6. CÓDIGO AST

Resumen de las clases utilizadas

7. FUNCIONES AUXILIARES

Resumen de las funciones auxiliares que se hayan codificado.

Funciones matemáticas

Funciones alfanuméricas

Etc.

8. MODO DE OBTENCIÓN DEL INTÉRPRETE

o Nombre y descripción de cada fichero utilizado

o Descripción del fichero ***makefile***

9. MODO DE EJECUCIÓN DEL INTÉRPRETE

o Interactiva

o A partir de un fichero

10. EJEMPLOS

o **IMPORTANTE**

Al menos se deben proporcionar dos ejemplos originales.

Se valorará la cantidad, originalidad y complejidad de los ejemplos propuestos.

o También se puede incluir el ejemplo propuesto por el profesor.

11. CONCLUSIONES

o Reflexión sobre el trabajo realizado.

o Puntos fuertes y puntos débiles del intérprete desarrollado.

12. BIBLIOGRAFÍA O REFERENCIAS WEB

o Se recomienda consultar el documento elaborado por el personal de la biblioteca de la Universidad de Córdoba

13. ANEXOS

o Se podrían incluir aquellos anexos que se consideren oportunos para mejora la calidad de la documentación