
# Diunisio: Manual técnico

**Andrés Felipe De Orcajo Vélez**
**Jorge Eduardo Ortiz Triviño**

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2017

# Índice

# 1. Definición de la gramática de Diunisio

⟨algoritmo⟩ ⊨ ALGORITMO ⟨identificador⟩ ⟨pars⟩ : ⟨bloque⟩ .

⟨pars⟩ ⊨ ( ⟨lista_ids⟩ ) | $\epsilon$

⟨lista_ids⟩ ⊨ ⟨identificador⟩ , ⟨lista_ids⟩ | ⟨identificador⟩

⟨exp_simple⟩ ⊨ ⟨op1⟩ ⟨termino⟩ ⟨terminos⟩ | ( ⟨exp_simple⟩ )

⟨terminos⟩ ⊨ ⟨op2⟩ ⟨termino⟩ ⟨terminos⟩ | $\epsilon$

⟨op1⟩ ⊨ + | - | $\epsilon$

⟨op2⟩ ⊨ + | - | ||

⟨expresion⟩ ⊨ ( ⟨expresion⟩ ) | ⟨exp_simple⟩ ⟨op3⟩ ⟨exp_simple⟩ | ⟨exp_simple⟩
| ! ⟨expresion⟩

⟨op3⟩ ⊨ = | != | <= | >= | < | >

⟨variable⟩ ⊨ ⟨identificador⟩ ⟨conjunto⟩ | ⟨identificador⟩

⟨termino⟩ ⊨ ⟨factor⟩ ⟨terms⟩ | ! ⟨termino⟩ | ⟨factor⟩ | ( ⟨termino⟩ )

⟨terms⟩ ⊨ ⟨op4⟩ ⟨factor⟩ ⟨terms⟩ | $\epsilon$

⟨op4⟩ ⊨ * | / | % | && | || | ^

⟨factor⟩ ⊨ ⟨entero⟩ | ⟨real⟩ | ⟨cadena⟩ | ⟨complejo⟩ | nulo | ⟨variable⟩
| ⟨identificador⟩ ⟨lista_parsv⟩ | ⟨identificador⟩ ⟨factores⟩
| ⟨identificador⟩ | ! ⟨factor⟩ | verdadero | falso | ⟨conjunto⟩
| ( ⟨expresion⟩ )

⟨factores⟩ ⊨ [ ⟨factor⟩ ] ⟨factores⟩ | [ ⟨factor⟩ ]

⟨lista_parsv⟩ ⊨ ( ⟨lista⟩ ) | ( )

⟨lista⟩ ⊨ ⟨lista⟩ , ⟨lista2⟩ | ⟨lista2⟩

⟨lista2⟩ ⊨ ⟨expresion⟩ | ⟨variable⟩ | ⟨identificador⟩

⟨conjunto⟩ ⊨ { ⟨expresiones⟩ } | { }

⟨expresiones⟩ ⊨ ⟨expresion⟩ , ⟨expresiones⟩ | ⟨expresion⟩

⟨tipo⟩ ⊨ entero | decimal | cadena | booleano | matriz | vector

⟨bloque⟩ ⊨ { ⟨sec_proposiciones⟩ } | { }

⟨sec_proposiciones⟩ ⊨ ⟨proposicion⟩ ⟨sec_proposiciones⟩ | ⟨proposicion⟩

⟨proposicion⟩ ⊨ retornar ⟨expresion⟩ ; | ⟨fun_senten⟩ | ⟨proc_senten⟩
| ⟨si_senten⟩ | ⟨seleccionar_senten⟩ | ⟨mientras_senten⟩
| ⟨para_senten⟩ | ⟨hacer_mientras_senten⟩ | ⟨asignacion⟩ ;
| ⟨identificador⟩ ⟨lista_parsv⟩ ; | { ⟨sec_proposiciones⟩ }

⟨asignacion⟩ ⊨ ⟨identificador⟩ = ⟨expresion⟩ | ⟨identificador⟩ = ⟨conjunto⟩

⟨si_senten⟩ ⊨ si ⟨bloque_condicional⟩ ⟨si_nos⟩ ⟨si_no⟩

⟨si_nos⟩ ⊨ si_no si ⟨bloque_condicional⟩ ⟨si_nos⟩ | $\epsilon$

⟨si_no⟩ ⊨ si_no ⟨entonces⟩ ⟨bloque⟩ | $\epsilon$

⟨entonces⟩ ⊨ entonces | $\epsilon$

⟨bloque_condicional⟩ ⊨ ⟨expresion⟩ ⟨entonces⟩ ⟨bloque⟩

⟨mientras_senten⟩ ⊨ mientras ⟨bloque_condicional⟩

⟨hacer_mientras_senten⟩ ⊨ hacer ⟨bloque⟩ mientras ⟨expresion⟩

⟨seleccionar_senten⟩ ⊨ seleccionar ⟨identificador⟩ { ⟨casos⟩ }

⟨casos⟩ ⊨ caso ⟨expresion⟩ : ⟨sec_proposiciones⟩ ⟨romper⟩ ⟨casos⟩
| defecto : ⟨sec_proposiciones⟩

⟨romper⟩ ⊨ romper ; | $\epsilon$

⟨para_senten⟩ ⊨ para ⟨asignaciones⟩ ; ⟨expresion⟩ ; ⟨asignaciones⟩ ⟨bloque⟩
| para ( ⟨asignaciones⟩ ; ⟨expresion⟩ ; ⟨asignaciones⟩ ) ⟨bloque⟩

⟨asignaciones⟩ ⊨ ⟨asignacion⟩ , ⟨asignaciones⟩ | ⟨asignacion⟩

⟨fun_senten⟩ ⊨ def ⟨tipo⟩ ⟨identificador⟩ ( ⟨lista_ids⟩ ) ⟨bloque⟩

⟨proc_senten⟩ ⊨ def ⟨identificador⟩ ( ⟨lista_ids⟩ ) ⟨bloque⟩

⟨letras⟩ ⊨ a . . . z ⟨letras⟩ | A . . . Z ⟨letras⟩ | A . . . Z | a . . . z

⟨numeros⟩ ⊨ 0 . . . 9 ⟨numeros⟩ | 0 . . . 9

⟨identificador⟩ ⊨ ⟨letras⟩ ⟨numeros⟩ | ⟨letras⟩

⟨real⟩ ⊨ ⟨num⟩ . ⟨numeros⟩ ⟨exponente⟩ | ⟨numeros⟩

⟨num⟩ ⊨ ⟨numeros⟩ | $\epsilon$

⟨exponente⟩ ⊨ ⟨exp⟩ ⟨op0⟩ ⟨numeros⟩ | $\epsilon$

⟨exp⟩ ⊨ e | E

⟨complejo⟩ ⊨ ⟨real⟩ ⟨op⟩ ⟨real⟩ i | ⟨real⟩ ⟨op⟩ i

⟨op⟩ ⊨ + | -

⟨op0⟩ ⊨ + | - | $\epsilon$

# 2. ANTLR

El lenguaje fue desarrollado en ANTLR[2], definiendo el léxico y la gramática en un archivo propio de ANTLR (.g4). Luego se generó el reconocedor, que creaba todas las clases bases en Java. Se sobrescribió el visitor donde se definía la semántica del lenguaje. Finalmente se creó una clase Main que es la que permite ejecutar algoritmos dentro del lenguaje.

## 2.1. Tokens del lenguaje

```
COMENTARIO : ('#' ~[\r\n]*  | '/*' .*? '*/') -> skip;
ALGORITMO : 'ALGORITMO';
TERMINA : '.';
ENTONCES : 'entonces';
O : '||';
Y : '&&';
IGUAL : '==';
DIFERENTE : '!=';
MAYOR : '>';
MENOR : '<';
MAY_IGUAL : '>=';
MEN_IGUAL : '<=';
SUMA : '+';
MENOS : '-';
MULT : '*';
DIV : '/';
MOD : '%';
POTENCIA : '^';
NO : '!';
DEF : 'def';
RETORNAR : 'retornar';
INT : 'entero';
FLOAT : 'decimal';
STRING : 'cadena';
BOOL : 'booleano';
MATRIZ : 'matriz';
VECTOR : 'vector';
PCOMA : ';';
ASIGNAR : '=';
PAREN_AP : '(';
PAREN_CI : ')';
LLAVEIZ : '{';
LLAVEDE : '}';
ANGIZ : '[';
ANGDE : ']';
COMA : ',';
DOSPUNTOS : ':';
VERDADERO : 'verdadero';
FALSO : 'falso';
NULO : 'nulo';
SI : 'si';
SI_NO : 'si_no';
MIENTRAS : 'mientras';
SELECCIONAR : 'seleccionar';
CASO: 'caso';
ROMPER: 'romper';
HACER: 'hacer';
PARA : 'para';
DEFECTO : 'defecto';
IDENTIFICADOR : [a-zA-Z_] [a-zA-Z_0-9]*;
ENTERO : [0-9]+;
REAL : [0-9]* '.' [0-9]* ([eE] [+-]? [0-9]+)?;
COMPLEJO : (ENTERO|REAL) [+|-] (ENTERO|REAL)? 'i';
CADENA : '"' (~["\r\n] | '""')* '"';
ESPACIO : [ \t\r\n] -> skip;
OTRO : .;
```

## 2.2. Gramática (en notación E-BNF, Extended Backus–Naur Form)

```
algoritmo
 : ALGORITMO IDENTIFICADOR (PAREN_AP lista_ids PAREN_CI)? DOSPUNTOS bloque TERMINA
 ;

lista_ids
```

```
 : IDENTIFICADOR (COMA IDENTIFICADOR)*
 |
 ;

exp_simple
 : PAREN_AP exp_simple PAREN_CI
 | (op=(SUMA | MENOS))? termino (op2=(SUMA | MENOS | O) termino)*
 | termino
 ;

expresion
 : PAREN_AP expresion PAREN_CI
 | exp_simple op=(IGUAL | DIFERENTE | MEN_IGUAL | MAY_IGUAL | MENOR | MAYOR) exp_simple
 | PAREN_AP exp_simple PAREN_CI
 | exp_simple
 | NO expresion
 ;

variable
 : IDENTIFICADOR conjunto
 | IDENTIFICADOR
 ;

termino
 : PAREN_AP termino PAREN_CI
 | factor (op=(MULT | DIV | MOD | Y | O | POTENCIA) factor)*
 | NO termino
 | factor
 ;

factor
 : ENTERO
 | REAL
 | CADENA
 | COMPLEJO
 | NULO
 | variable
 | IDENTIFICADOR lista_parsv?
 | IDENTIFICADOR (ANGIZ factor ANGDE)+
 | IDENTIFICADOR
 | NO factor
 | VERDADERO
 | FALSO
 | conjunto
 | PAREN_AP expresion PAREN_CI
 ;

lista_parsv
 : PAREN_AP (expresion | variable | IDENTIFICADOR)
   (COMA (expresion | variable | IDENTIFICADOR ))* PAREN_CI
 | PAREN_AP PAREN_CI
 ;

conjunto
 : LLAVEIZ (expresion (COMA expresion)*)? LLAVEDE
 ;

tipo
 : INT | FLOAT | STRING | BOOL | MATRIZ | VECTOR;

bloque
 : LLAVEIZ LLAVEDE
 | LLAVEIZ sec_proposiciones LLAVEDE
 ;

sec_proposiciones
 : (proposicion)* proposicion
 ;

proposicion
 : RETORNAR expresion PCOMA
 | fun_senten
```

```
 | proc_senten
 | si_senten
 | seleccionar_senten
 | mientras_senten
 | para_senten
 | hacer_mientras_senten
 | asignacion PCOMA
 | IDENTIFICADOR lista_parsv PCOMA
 | LLAVEIZ sec_proposiciones LLAVEDE
 | OTRO {System.err.println("Caracter desconocido: " + $OTRO.text);}
 ;

asignacion
 : IDENTIFICADOR ASIGNAR expresion  #asigNum
 | IDENTIFICADOR ASIGNAR conjunto   #asigVec
 ;

si_senten
 : SI bloque_condicional (SI_NO SI bloque_condicional)* (SI_NO ENTONCES? bloque)?
 ;

bloque_condicional
 : expresion ENTONCES? bloque
 ;

mientras_senten
 : MIENTRAS bloque_condicional
 ;

hacer_mientras_senten
 : HACER bloque MIENTRAS expresion
 ;

seleccionar_senten
 : SELECCIONAR IDENTIFICADOR LLAVEIZ casos LLAVEDE
 ;

casos
 : CASO expresion DOSPUNTOS sec_proposiciones (ROMPER PCOMA)? casos     #casosGen
 | DEFECTO DOSPUNTOS sec_proposiciones                                  #casosDef
 ;

para_senten
 : PARA asignacion (COMA asignacion)* PCOMA expresion PCOMA asignacion
   (COMA asignacion)* bloque
 | PARA PAREN_AP asignacion (COMA asignacion)* PCOMA expresion PCOMA asignacion
   (COMA asignacion)* PAREN_CI bloque
 ;

fun_senten
 : DEF tipo IDENTIFICADOR PAREN_AP lista_ids PAREN_CI bloque
 ;

proc_senten
 : DEF IDENTIFICADOR PAREN_AP lista_ids PAREN_CI bloque
 ;
```

## 2.3. Clases de Java

- `Valor.java` en donde se definen los átomos del lenguaje, donde se almacena la información de cada dato.
- `EvalVisitor.java` en donde se sobrescriben los métodos de `DiunisioBaseVisitor.java` previamente generado a partir de la gramática en ANTLR.
- `Main.java` donde se ejecutan los algoritmos usando Diunisio.
- `Complex.java` en donde se definen los métodos para números complejos[3]
- `FunctionValor.java` donde se extiende de la clase `Valor.java` para definir funciones y procedimientos añadiendo más atributos y métodos.

## 2.4. Diagramas de sintaxis

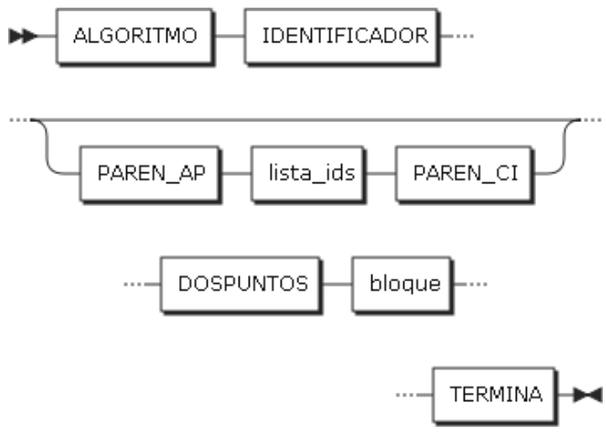


Figura 1: Algoritmo

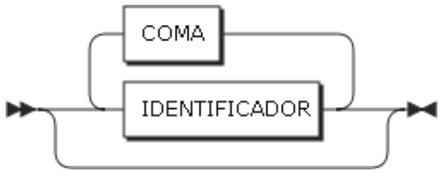


Figura 2: Lista de identificadores

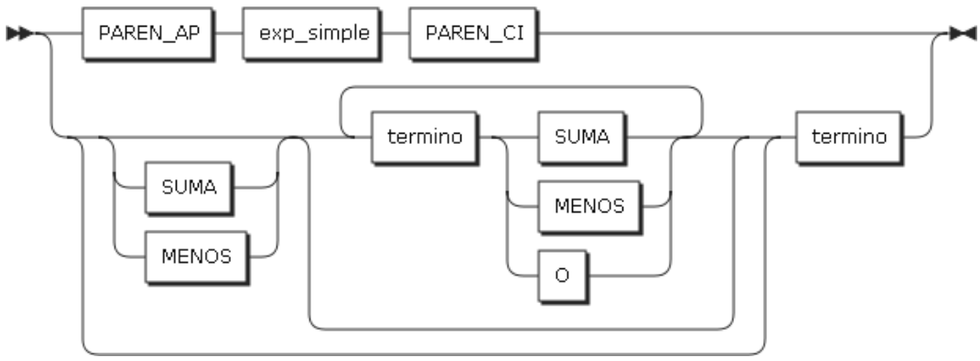


Figura 3: Expresión simple

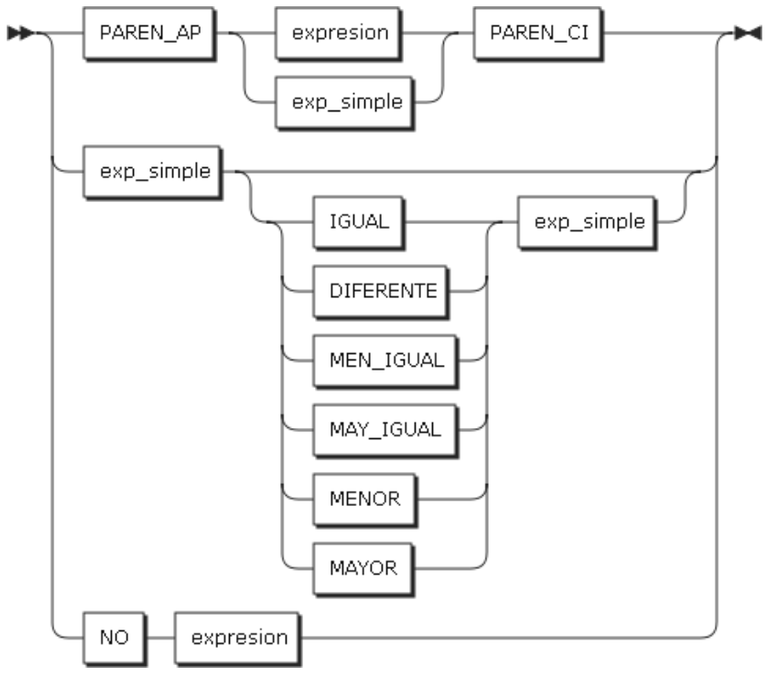


Figura 4: Expresión

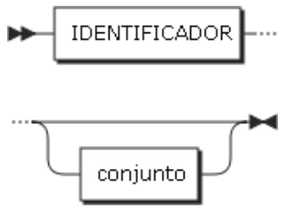


Figura 5: Variable

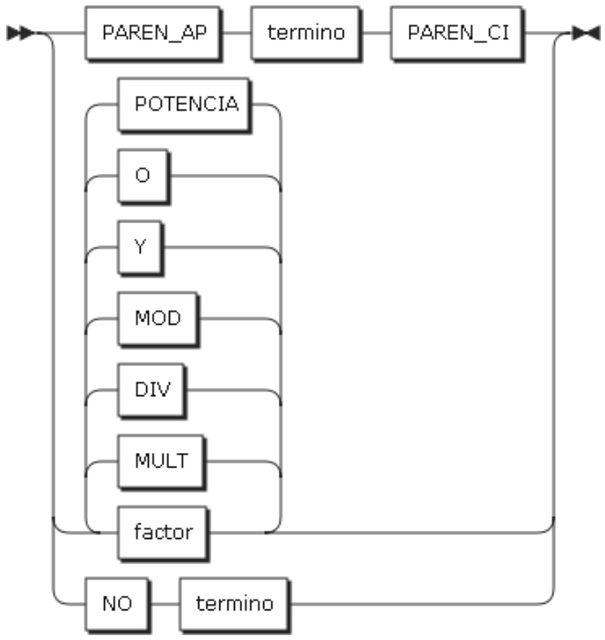


Figura 6: Término

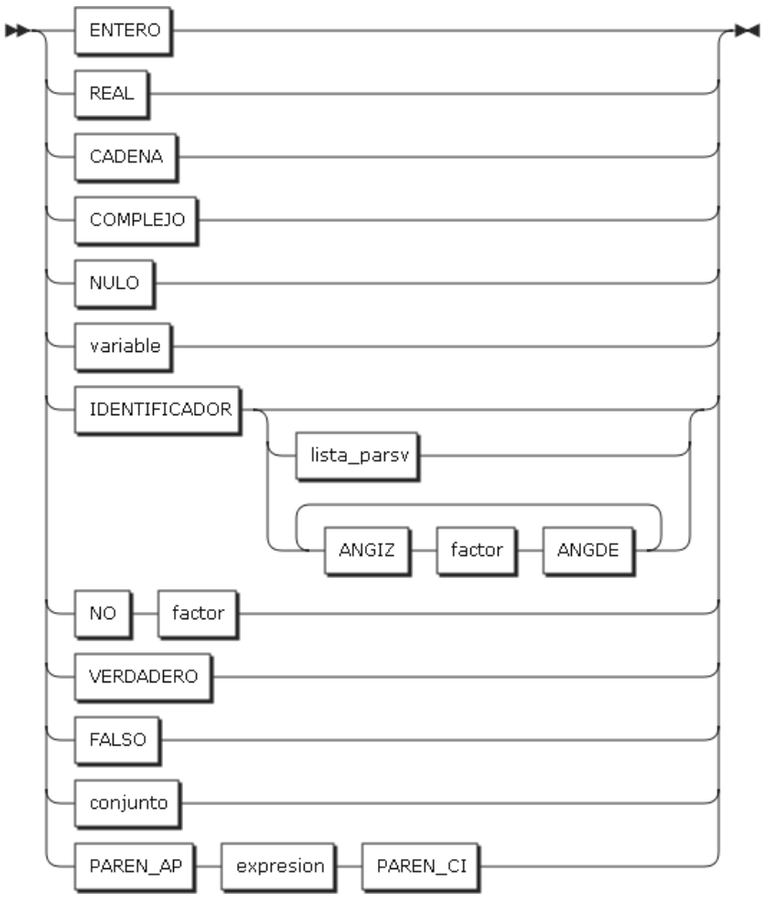


Figura 7: Factor

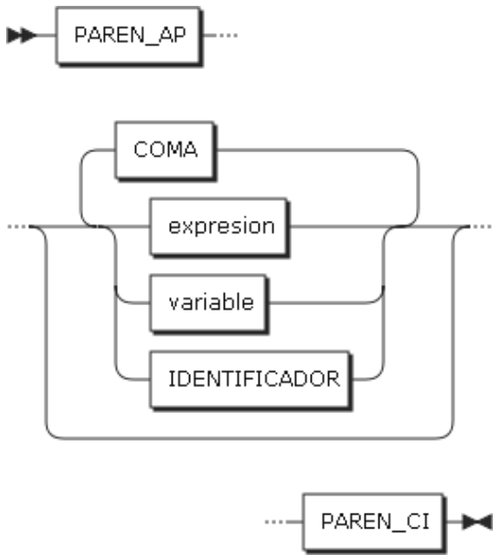


Figura 8: Lista de parámetros verdaderos

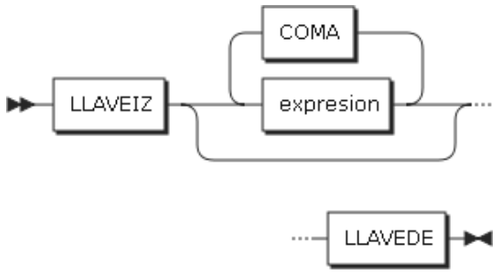


Figura 9: Conjunto

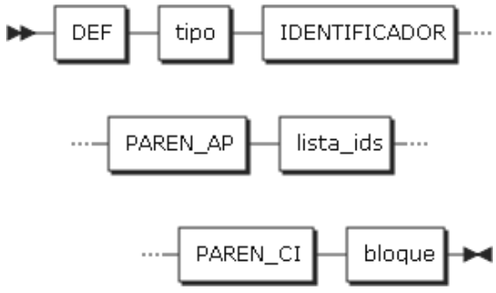


Figura 10: Fun_Senten (Definición de función)

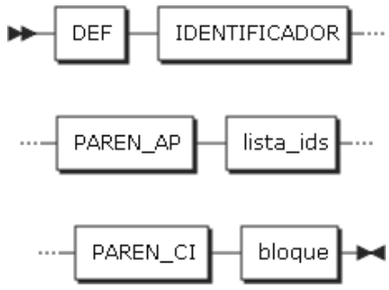


Figura 11: Proc_Senten (Definición de procedimiento)

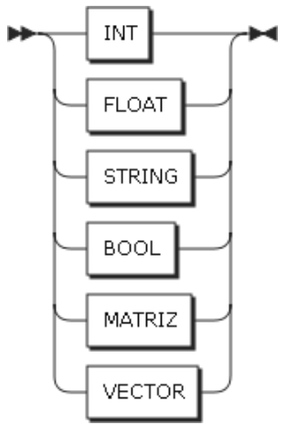


Figura 12: Tipo

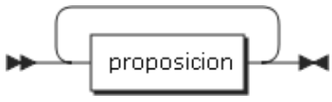


Figura 13: Secuencia de proposiciones

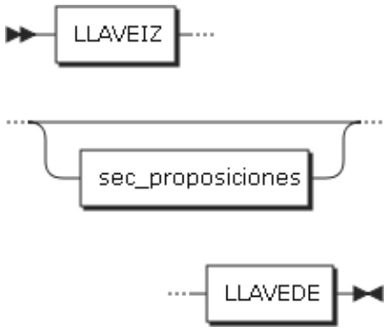


Figura 14: Bloque

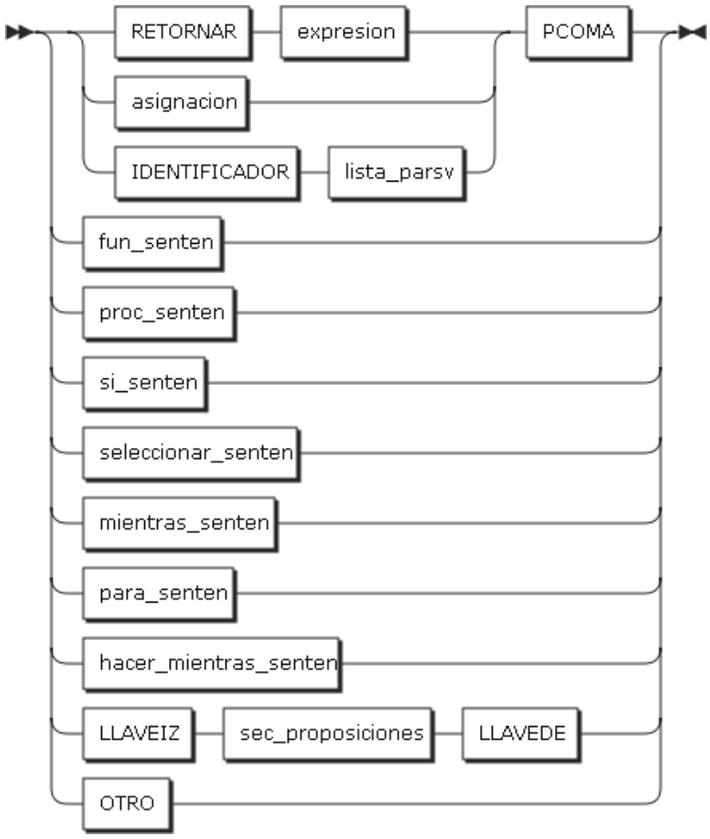


Figura 15: Proposición

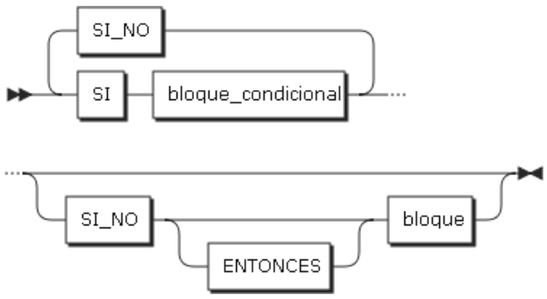


Figura 16: Si_Senten (Sentencia SI)

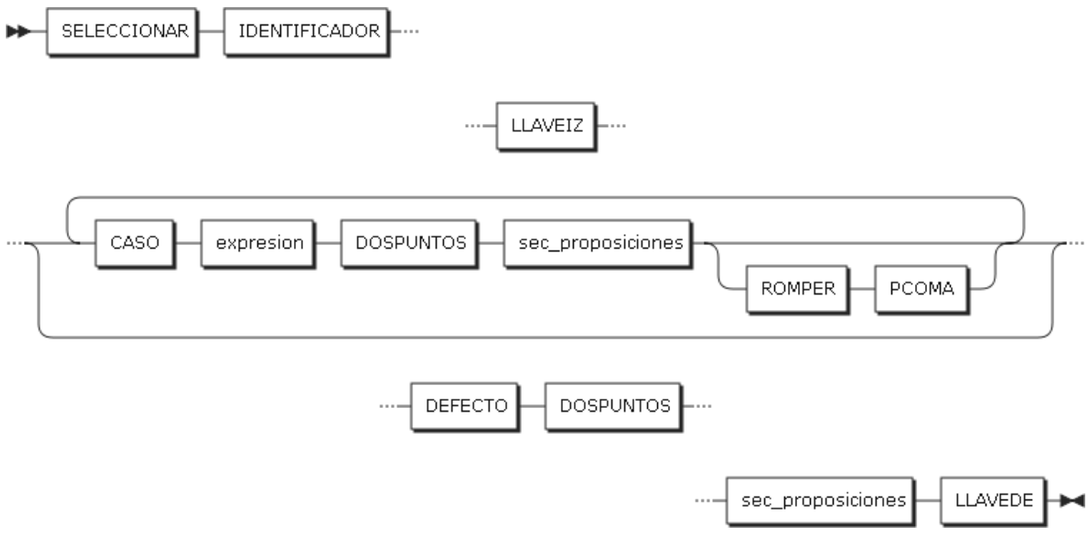


Figura 17: Seleccionar_Senten (Sentencia SELECCIONAR)


Figura 18: Mientras_Senten (Sentencia MIENTRAS)

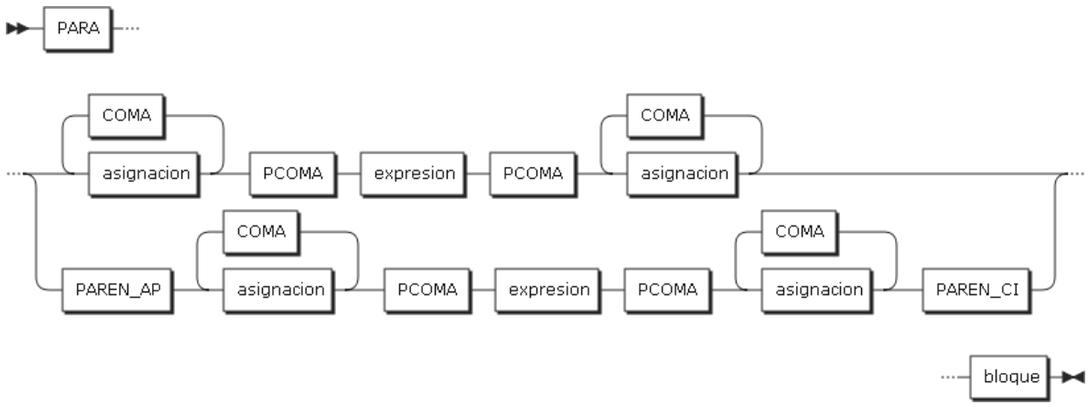


Figura 19: Para_Senten (Sentencia PARA)

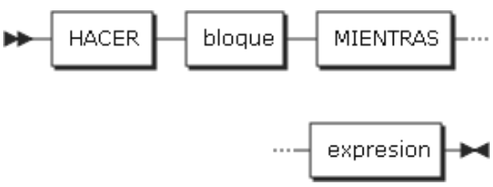


Figura 20: Hacer_Mientras_Senten (Sentencia HACER MIENTRAS)

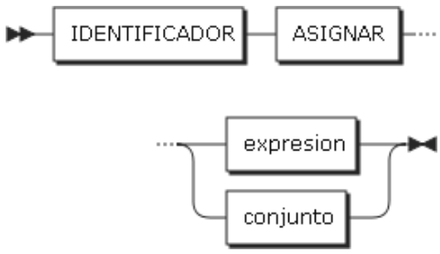


Figura 21: Asignación

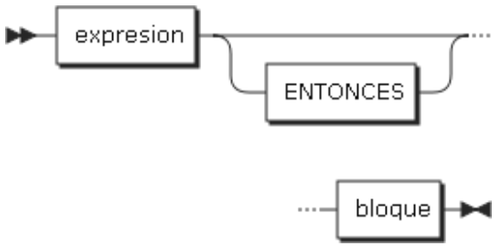


Figura 22: Bloque condicional

## 2.5. Generación de árbol de parseo

Para generar un árbol de parseo es necesario un algoritmo previamente diseñado con la sintaxis correcta. Se usa el comando `grun Diunisio algoritmo`, donde: `Diunisio` es la gramática y `algoritmo` es la producción inicial.

### 2.5.1. Pruebas de generación de árboles gráficos

Para generarlos gráficamente es necesario ejecutar desde el terminal o la consola el comando `grun` de ANTLR con el parámetro `-gui`, lo que generará un árbol gráfico.

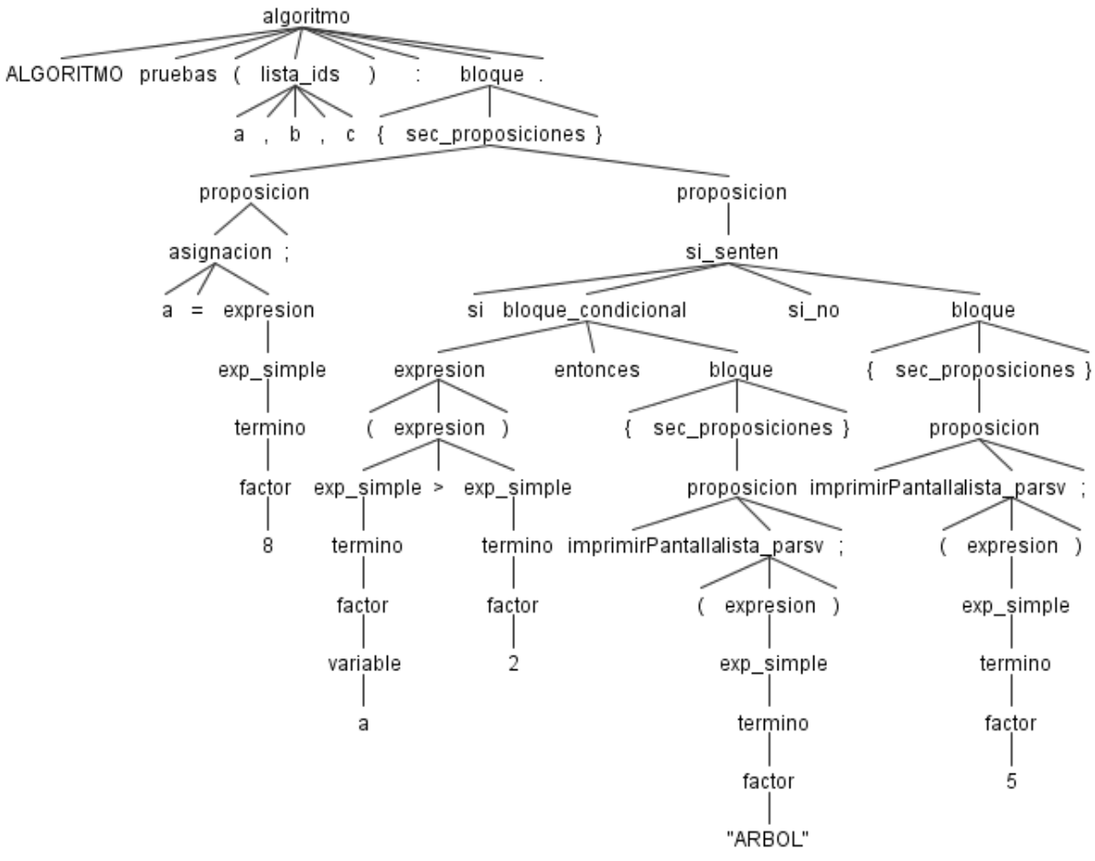


Figura 23: Bloque condicional

### 2.5.2. Pruebas de generación de árboles con caracteres

Se usa desde consola el comando `grun Diunisio algoritmo -tree` y luego se inserta el algoritmo para conocer su árbol de sintaxis. Por ejemplo:

```
(algoritmo ALGORITMO pruebas ( (lista_ids a , b , c) ) : (bloque { (sec_proposiciones
(proposicion (asignacion a = (expresion (exp_simple (termino (factor 8))))) ;) (propos
icion (si_senten si (bloque_condicional (expresion ( (expresion (exp_simple (termino (
factor (variable a)))) > (exp_simple (termino (factor 2)))) )) entonces (bloque { (sec
_proposiciones (proposicion imprimirPantalla (lista_parsv ( (expresion (exp_simple (te
rmino (factor "ARBOL")))) )) ;)) })) si_no (bloque { (sec_proposiciones (proposicion i
mprimirPantalla (lista_parsv ( (expresion (exp_simple (termino (factor 5)))) )) ;)) })
))) }) .)
```

## 3. Recomendaciones y Sugerencias

Para desarrollar un lenguaje de programación usando ANTLR, es necesario pensar en la funcionalidad del mismo junto con sus requerimientos. Desde un comienzo se debe definir una gramática correctamente para que en su implementación no se generen modificaciones ni alteraciones.También es necesario conocer el funcionamiento de ANTLR para aprovechar sus capacidades al máximo.
Es importante hacer pruebas con la gramática inicial para reconocer falencias tempranamente y corregirlas.

## Referencias


[1] Generación de los diagramas E-BNF: `http://www.bottlecaps.de/rr/ui`

[2] ANTLR (Versión 4.7) {software}. (2017). Obtenido de: `http://www.antlr.org/`

[3] jcomplexnumber {software}. (2016). Obtenido de: `https://github.com/abdulfatir/jcomplexnumber/`