

**Procesadores de Lenguajes**

**Analizador Sintáctico**

**Docente**: Cristian Prats

**Integrantes**:

Geronimo Ratcliffe

Javier Villagra

Santiago Russo

Fecha: 20/11/2013

Versión: 1.0

Tabla de Revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Descripción | Autor |
| 20/11/2013 | 1.0 | Creación del Documento | Santiago Russo |

Contents

[I: Introducción 7](#_Toc372722344)

[II: Definiciones de Reglas y Atributos 7](#_Toc372722345)

[2.1 Gramática en el Algoritmo 13](#_Toc372722346)

[III: Tabla de Símbolos 13](#_Toc372722347)

[3.1 Manejo de Recursividad 14](#_Toc372722348)

[3.2 Manejo del Contexto 14](#_Toc372722349)

[IV: Proceso De Implementación 14](#_Toc372722350)

[4.1 Mecanismo De Recuperación De Errores 14](#_Toc372722351)

[V: Pruebas 15](#_Toc372722352)

[VI: Cambios Realizados al Sintáctico 15](#_Toc372722353)

[VII: Conclusiones 15](#_Toc372722354)

[A: Anexo Diagramas de Clases 16](#_Toc372722355)

[A.1: Reglas y No Terminales 16](#_Toc372722356)

[A.1.1: Bloque 16](#_Toc372722357)

[A.1.2: Decl 18](#_Toc372722358)

[A.1.3 Exp 19](#_Toc372722359)

[A.1.4: Func 20](#_Toc372722360)

[A.2: Nodos y Estructuras 21](#_Toc372722361)

[A.2.1 Bloque 22](#_Toc372722362)

[A.2.3 Decl 23](#_Toc372722363)

[A.2.3 EXP 24](#_Toc372722364)

[A.2.4: FUNC 25](#_Toc372722365)

# I: Introducción

El siguiente documento explicara a grandes rasgos el proceso de implementación del Analizador Sintáctico utilizado. Adjunto se encontrara un CD con el código fuente, ejecutable y pruebas.

Se disculpa la breve documentación del código, comentarios, etc.

# II: Definiciones de Reglas y Atributos

La definición de Reglas y atributos que se utilizó, a modo de base, para el trabajo es la siguiente:

|  |  |  |
| --- | --- | --- |
| Producción | Regla Semántica | Tipo de Nodo |
| S' -> <S> | S'.arbol = S.arbol |  |
| S -> <GLOBALES> <BLOQUE> | if(!Tabla.existeContexto("main")) then semanticError("Error, falta el main");S.arbol = crearnodo(GLOBALES.arbol, BLOQUE.arbol); |  |
| GLOBALES -> <DECGL> | GLOBALES.arbol = new Arbol (DECGL.lista); |  |
| GLOBALES -> λ | GLOBALES.arbol = new Arbol(); |  |
| BLOQUE -> <FP> <BLOQUE'> | BLOQUE.arbol = BLOQUE'.arbol;BLOQUE.arbol.add(FP.arbol); |  |
| BLOQUE -> λ | BLOQUE.arbol = new NodoBloque(); |  |
| FP -> FUNCTION ID PARENTESIS\_APERTURA <PARAM> PARENTESIS\_CIERRE DEF\_TIPO TIPO ENDLINE <BLOQUE1> END\_FUNC <EXP> ENDLINE | if(TIPO.tipo != EXP.tipo) then typeError() else (if(Tabla.existe(ID.lexval) then semanticError("Ya existe la funcion ID.lexval") else (FP.arbol = crearnodoFunc(ID.lexval, PARAM.arbol, BLOQUE1.decl, BOQUE1.bloque, EXP.arbol);Tabla.addFuncGlobal(ID.lexval, PARAM.arbol, TIPO.tipo) Tabla.addParam(IPARAM.arbol); | 1) Seteas el Contexto |
| FP -> PROCEDURE ID PARENTESIS\_APERTURA <PARAM> PARENTESIS\_CIERRE ENDLINE <BLOQUE1> END\_PROC ENDLINE | if(Tabla.existe(ID.lexval) then semanticError("Ya existe la funcion ID.lexval") else (FP.arbol = crearnodoProc(ID.lexval, new NodoParam(PARAM.lista), new NodoDecl(BLOQUE1.decl), NewNodoBloque(BOQUE1.bloque)); Tabla.addFuncGlobal(ID.lexval, PARAM.arbol)Tabla.cambioContexto(ID.lexval)); Tabla.addParam(IPARAM.arbol); |  |
| BLOQUE1 -> <DECL> BEGIN <BLOQUE2> | BLOQUE1.decl = DECL.lista; BLOQUE1.bloque = BLOQUE2.lista |  |
| BLOQUE1 -> λ | BLOQUE1.decl = newArrayList<HojaDecl>(); BLOQUE1.bloque = new ArrayList<Sentacias>(); |  |
| BLOQUE2 -> <LINEA> ENDLINE <BLOQUE2'> | BLOQUE2.lista = BLOQUE2'.lista; BLOQUE2.lista.add(LINEA.arbol); |  |
| BLOQUE2 -> λ | BLOQUE2.lista = newArrayList<Sentencias>(); |  |
| LINEA -> ID <ASIG> | if(Tabla.existe(ID.lexema) then (if(ASIG.tipoNodo == Acceso) then (if(buscarTabla(ID.lexema).tipo != ASIG.tipo) then typeError() else LINEA.arbol = crearnodoAsigAcceso(ID.lexema, ASIG.arbol)) else if(ASIG.tipoNodo == Array) then ( LINEA.arbol = crearnodoAsigAccesoArray(ID.lexema, ASIG.indice, ASIG.arbol) if(buscarTabla(ID.lexema).tipo != ASIG.tipo) then typeError();) else (LINEA.arbol = crearNodoAsigLlamada(ID.lexema, ASIG.arbol) if(buscarFuncProcTabla(ID.lexema).chequeoParam(ASIG.arbol); |  |
| LINEA -> IF <EXP> THEN <BLOQUE2> <BLOQUESI> END\_IF | if(EXP.tipo != boolean) then typeError() else (LINEA.arbol = crearnodoIf(EXP.arbol, BLOQUE2.lista, BLOQUESI.lista)); |  |
| LINEA -> WHILE <EXP> DO <BLOQUE2> END\_WHILE | if(EXP.tipo != boolean) then typeError() else (LINEA.arbol = crearnodoWhile(EXP.arbol, BLOQUE2.lista)); |  |
| LINEA -> <FUNC> | LINEA.arbol = FUNC.arbol |  |
| ASIG -> PARENTESIS\_APERTURA <PASAJE> PARENTESIS\_CIERRE | ASIG.tipoNodo == Llamada; ASIG.arbol = PASAJE.arbol; ASIG.tipos = PASAJE.tipos; | tipos es una ArrayList de tipos |
| ASIG -> ASIGNACION <EXP> | ASIG.tipoNodo == Acceso; ASIG.arbol = EXP.arbol; ASIG.tipo = EXP.tipo |  |
| ASIG -> CORCHETE\_APERTURA <EXP'> CORCHETE\_CIERRE ASIGNACION <EXP''> | if(EXP'.tipo != integer) then typeError("Array index not numeric") else ASIG.tipoNodo == Array; ASIG.indice = EXP'.arbol; ASIG.arbol = EXP''.arbol; ASIG.tipo = EXP''.tipo; |  |
| BLOQUESI -> ELSE <BLOQUE2> | BLOQUESI.lista = BLOQUE2.lista | es ArrayList<Sentencias> |
| BLOQUESI -> λ | BLOQUESI.lista = null; |  |
| FUNC -> READ ID <IDAUX> | if(IDAUX.isArray) then FUNC.arbol = new nodoRead(ID.lexval, IDAUX.expIndice) else FUNC.arbol = new nodoRead(ID.lexval) |  |
| FUNC -> SHOW <MOSTRAR> | FUNC.arbol = new nodoShow(SHOW.lexval,MOSTRAR.lista) // Despues veo si SHOW.lexval es ShowLn --> agrego \n |  |
| PARAM -> <TIPOPARAM> ID DEF\_TIPO TIPO <PARAM1> | if(TIPOPARAM.byref) then PARAM1.lista.addFirst(newHojaParam(true,ID.lexval, TIPO.tipo)) ELSE PARAM1.lista.addFirst(newHojaParam(false,ID.lexval, TIPO.tipo)); PARAM1.lista = PARAM1'.lista; Tabla.addParam(); |  |
| PARAM -> λ | PARAM1.lista = new ArrayList<HojaParam>(); |  |
| PARAM1 -> COMMA <TIPOPARAM> ID DEF\_TIPO TIPO <PARAM1'> | if(TIPOPARAM.byref) then PARAM1.lista.addFirst(newHojaParam(true,ID.lexval, TIPO.tipo)) ELSE PARAM1.lista.addFirst(newHojaParam(false,ID.lexval, TIPO.tipo)); PARAM1.lista = PARAM1'.lista; Tabla.addParam(); |  |
| PARAM1 -> λ | PARAM1.lista = new ArrayList<HojaParam>(); |  |
| TIPOPARAM -> BY | if(BY.lexval == ByRef) then TIPOPARAM.byRef = true else TIPOPARAM.byRef = false |  |
| TIPOPARAM -> λ | TIPOPARAM.byref = false; |  |
| DECGL -> <VARG> <DECGL'> | DECGL.lista = DECGL'.lista; DECGL.lista.addFirst(VARG.lista); | NodoDecl |
| DECGL -> <CONSTS> <DECGL'> | DECGL.lista = DECGL'.lista; DECGL.lista.addFirst(CONSTS.lista); | NodoDecl |
| DECGL -> λ | DECGL.lista = new ArrayList<HojaDecl>(); | NodoDecl |
| DECL -> <VARS> <DECL'> | DECL.lista = DECL'.lista; DECL.lista.add(VARS.lista); | NodoDecl |
| DECL -> <CONSTS> <DECL'> | DECL.lista = DECL'.lista; DECL.lista.add(CONSTS.lista); | NodoDecl |
| DECL -> λ | DECL.lista = new ArrayList<HojaDecl>(); | NodoDecl |
| CONSTS -> CONST ID DEF\_TIPO <CONST1> ENDLINE | CONSTS.lista = CONST1.lista; CONSTS.lista.addFirst(new HojaConst(ID.lexval, CONST1.tipo, CONST1.valor)); Tabla.addConst(); | NodoConst extends NodoDeclarable |
| CONST1 -> TIPO CONST\_IGUAL BOOLEANO <CONST2> | if(TIPO.tipo == boolean) then (CONST1.tipo = boolean; CONST1.valor = BOOLEANO.lexema; CONST1.lista = CONST2.lista;) else (typeError("Type Error defining boolean constant"); | NodoConst extends NodoDeclarable |
| CONST1 -> TIPO CONST\_IGUAL NUMERO <CONST2> | if( TIPO.tipo == integer) then (CONST1.tipo = integer; CONST1.valor = NUMERO.lexema; CONST1.lista = CONST2.lista;) else (typeError("Type Error defining integer constant")); | NodoConst extends NodoDeclarable |
| CONST2 -> COMMA ID DEF\_TIPO <CONST1> | CONST2.lista = CONST1.lista; CONST2.lista.addFirst(ID.lexval, CONST1.tipo, CONST1.valor); Tabla.addConst(); | NodoConst extends NodoDeclarable |
| CONST2 -> λ | CONST2.lista = newArrayList<HojaConst>() | NodoConst extends NodoDeclarable |
| VARS -> VAR ID <VAR1> ENDLINE | VARS.lista = VAR1.lista; VARS.lista.addFirst(newHojaVar(ID.lexval,VAR1.tipo)); VARS.tipo = VAR1.tipo; Tabla.addVar | NodoVar extends NodoDeclarable |
| VAR1 -> COMMA ID <VAR1'> | VAR1.lista = VAR1'.lista; VAR1.lista.addFirst(newHojaVar(ID.lexval, VAR1'.tipo)); VAR1.tipo = VAR1'.tipo; Tabla.addVar | NodoVar extends NodoDeclarable |
| VAR1 -> DEF\_TIPO TIPO | VAR1.lista = newArrayList<VAR>(); VAR1.tipo = TIPO.tipo | NodoVar extends NodoDeclarable |
| VARG -> VAR ID <VARG1> ENDLINE | if(VARG1.isArray) then VARG1.lista.addFirst(newHojaArray(ID.lexval, VARG1.tamaño,VARG1.tipo)) else VARG1.lista.addFirst(newHojaVar(ID.lexval,VARG1.tipo)); VARG.lista = VARG1.lista; this.tipo = VARG1.tipo; Tabla.addGlobalesVar o Tabla.addArray | NodoVar extends NodoDeclarable |
| VARG1 -> COMMA ID <VARG1'> | if(VARG1'.isArray) then VARG1'.addFirst(newHojaArray(ID.lexval, VARG1'.tamaño, VARG1'.tipo)) else VARG1'.lista.addFirst(newHojaVar(ID.lexval,VARG1.tipo)); VARG1.lista = VARG1'.lista; VARG1.isArray = false; VARG1'.tipo = VARG1.tipo; Tabla.addGlobalesVar o Tabla.addArray | NodoVar extends NodoDeclarable |
| VARG1 -> DEF\_TIPO TIPO | VARG1.lista = newArrayList<VARG>(); VARG1.isArray = false; VARG1.tipo = TIPO.tipo; | NodoVar extends NodoDeclarable |
| VARG1 -> CORCHETE\_APERTURA NUMERO CORCHETE\_CIERRE <VARG1'> | VARG1.isArray = true; VARG1.tamaño = NUMERO.lexval; VARG1.tipo = VARG1'.tipo; VARG1.lista = VARG1'.lista | NodoVar extends NodoDeclarable |
| MOSTRAR -> CADENA <MOSTRAR1> | MOSTRAR1.lista<Mostrable>.addFirst(CADENA.lexval); MOSTRAR.lista = MOSTRAR1.lista; | Mostrable es una inter para EXP y Cadena |
| MOSTRAR -> <EXP> <MOSTRAR1> | MOSTRAR1.lista<Mostrable>.addFirst(EXP.arbol); MOSTRAR.lista = MOSTRAR1.lista | Si el EXP es un ID en MOSTRAR voy a buscar el valor de ese ID, si es num o bool lo convierto a String |
| MOSTRAR1 -> COMMA <MOSTRAR> | MOSTRAR1.lista = MOSTRAR.lista |  |
| MOSTRAR1 -> λ | MOSTRAR1.lista = newArrayList<Mostrable>(); |  |
| IDAUX -> CORCHETE\_APERTURA <EXP> CORCHETE\_CIERRE | if(EXP.tipo = Integer) then IDAUX.isArray = true; IDAUX.expIndice = EXP.arbol else typeError(); |  |
| IDAUX -> λ | IDAUX.isArray = false; |  |
| EXP -> <EXP'> OR <TERMBOOL> | EXP.arbol=crearNodoEXP(OR.lexval,EXP'.arbol,TERMBOOL.arbol);EXP.tipo = boolean; if (EXP.tipo!= Boolean || TERMBOOL.tipo != Boolean) then error(); | NodoExp |
| EXP -> <TERMBOOL> | EXP.arbol = TERMBOOL.arbol;EXP.tipo=TERMBOOL.tipo | NodoExp |
| TERMBOOL -> <TERMBOOL'> AND <MINITERMBOOL> | TERMBOOL.arbol = crearNodoEXP(AND.lexval,TERMBOOL'.arbol, MINITERMBOOL.arbol) | NodoExp |
| TERMBOOL -> <MINITERMBOOL> | TERMBOOL.arbol = MINITERMBOOL.arbol; TERMBOOL.tipo = MINITERMBOOL.tipo; | NodoExp |
| MINITERMBOOL -> NOT <MINITERMBOOL> | if(MINITERMBOOL.tipo != Boolean) then TypeError() else MINITERMBOOL.arbol = crearNodoNOT(NOT.lexval, FACTBOOL.arbol); MINITERMBOOL.tipo = Boolean; | NodoNot extends NodoDecl |
| MINITERMBOOL -> <FACTBOOL> | MINITERMBOOL.arbol = FACTBOOL.arbol; MINITERMBOOL.tipo = FACTBOOL.tipo; | NodoExp |
| FACTBOOL -> <EXPBOOL> | FACTBOOL.arbol = EXPBOOL.arbol; this.tipo = Boolean | NodoExp |
| FACTBOOL -> <EXP2> | FACTBOOL.arbol = EXP2.arbol; this.tipo = EXP2.tipo; | NodoExp |
| EXP2 -> <EXP2'> SUMA\_RESTA <TERM> | if(EXP2'.tipo != Intiger o TERM.tipo != Intiger) then typeError() else EXP2.arbol=crearNodoEXP(SUMA\_RESTA.lexval,EXP2'.arbol,TERM.arbol); EXP2.tipo = Intiger; | NodoExp |
| EXP2 -> <TERM> | EXP2.arbol = TERM.arbol; EXP2.tipo = TERM.tipo | NodoExp |
| TERM -> <TERM'> MULT\_DIV <FACT> | if(TERM.tipo != Integer || FACT.tipo != Integer) then typeError() else TERM.arbol=crearNodoEXP(MULT\_DIV.lexval,TERM'.arbol,FACT.arbol); TERM.tipo = Integer | NodoExp |
| TERM -> <FACT> | TERM.arbol = FACT.arbol; TERM.tipo = FACT.tipo; | NodoExp |
| FACT -> PARENTESIS\_APERTURA <EXP> PARENTESIS\_CIERRE | FACT.NodoExp = EXP.arbol; FACT.tipo = EXP.tipo | NodoExp |
| FACT -> ID <ID1> | if(ID1.tipoNodo == Acceso) then FACT.arbol = crearnodoAcceso(ID.lexema) else if(ID1.tipoNodo == Array) then FACT.arbol = crearnodoAccesoArray(ID.lexema, ID1.arbol // seria el indice) else if(tipoParam(ID.lexema,ID1,arbol) then FACT.arbol = crearNodoLlamada(ID.lexema, ID1.arbol) else typeError; FACT.tipo = tabla.buscarTipo(ID); | NodoExp |
| FACT -> BOOLEANO | FACT.NodoExp = crearHoja(BOOLEANO.lexval); FACT.tipo = Boolean; | NodoExp |
| FACT -> NUMERO | FACT.NodoExp = crearHoja(NUMERO.lexval); FACT.tipo = Integer; | NodoExp |
| EXPBOOL -> <EXP2'> OPERADOR\_BOOLEANO <EXP2''> | if(EXP2'.tipo != EXP2''.tipo) then typeError() else (EXPBOOL.NodoExpBool= crearNodoEXPBOOL(OPERADOR\_BOOLEANO.lexval,EXP2'.arbol,EXP2''.arbol); EXPBOOL.tipo = Boolean | NodoExpBool |
| EXPBOOL -> <EXP2'> OPERADOR\_BOOLEANO\_E <EXP2''> | if(EXP2'.tipo != integer || EXP2''.tipo != integer) then typeError() else EXPBOOL.NodoExpBool = crearNodoEXPBOOL(OPERADOR\_BOOLEANO\_E.lexval,EXP2'.arbol,EXP2''.arbol); EXPBOOL.tipo = Boolean; | NodoExpBool |
| ID1 -> CORCHETE\_APERTURA <EXP> CORCHETE\_CIERRE | if(EXP.tipo != Integer) then typeError else ID1.tipoNodo = Array; ID1.NodoExp = EXP.getArbol(); | NodoExp |
| ID1 -> PARENTESIS\_APERTURA <PASAJE> PARENTESIS\_CIERRE | ID1.tipoNodo = Llamada; ID1.NodoPasaje = PASAJE.NodoPasaje | NodoPasaje |
| ID1 -> λ | ID1.tipoNodo = Acceso |  |
| PASAJE -> <EXP> <PASAJE1> | PASAJE.NodoPasaje = PASAJE1.NodoPasaje; PASAJE1.NodoPasaje.addFirst(EXP.NodoExp); | NodoPasaje |
| PASAJE -> λ | PASAJE.NodoPasaje = newListaPasaje(); | NodoPasaje |
| PASAJE1 -> COMMA <EXP> <PASAJE1'> | PASAJE1.NodoPasaje = PASAJE1'.NodoPasaje; PASAJE1.NodoPasaje.addFirst(EXP.NodoExp); | NodoPasaje |
| PASAJE1 -> λ | PASAJE1.NodoPasaje = newListaPasaje(); | NodoPasaje |

Como modo de ayuda se han introducido los atributos de cada Producción. Hay que advertir que algunos de los atributos han quedado obsoletos y han sido reemplazados.

|  |  |  |
| --- | --- | --- |
| Producción | Atributos | Paquete |
| S' | arbol | - |
| S | arbol | - |
| GLOBALES | lista | Bloque |
| BLOQUE | lista | Bloque |
| FP | arbol; Agrega a la Tabla y cambia contexto | Func |
| BLOQUE1 | decl | Bloque |
| bloque |
| BLOQUE2 | lista | Bloque |
| LINEA | arbol | Bloque |
| ASIG | arbol | Bloque |
| tipoNodo |
| indice |
| tipo |
| BLOQUESI | arbol | Bloque |
| FUNC | arbol | Func |
| PARAM | lista | Func |
| PARAM1 | lista | Func |
| TIPOPARAM | byRef | Func |
| DECGL | lista | Decl |
| Usa Tabla.addGlobales | Decl |
| DECL | lista | Decl |
| Usa Tabla.add | Decl |
| CONSTS | lista | Decl |
| CONST1 | tipo | Decl |
| valor | Decl |
| lista | Decl |
| CONST2 | lista | Decl |
| VARS | lista | Decl |
| VAR1 | lista | Decl |
| VARG | lista | Decl |
| VARG1 | lista / tipo | Decl |
| isArray //booleano | Decl |
| tamaño //para el arreglo | Decl |
| MOSTRAR | lista | Bloque |
| MOSTRAR1 | lista | Bloque |
| IDAUX | expIndice | Bloque |
| isArray //booleano |
| EXP | arbol; Tipo | Exp |
| TERMBOOL | arbol; Tipo | Exp |
| MINITERMBOOL | arbol; Tipo | Exp |
| FACTBOOL | arbol; Tipo | Exp |
| EXP2 | arbol; Tipo | Exp |
| TERM | arbol; Tipo | Exp |
| FACT | arbol; Tipo | Exp |
| EXPBOOL | arbol; Tipo | Exp |
| ID1 | arbol //puede venir de EXP.arbol o PASAJE.lista | Exp |
| tipoNodo / ListaTipos |
| PASAJE | nodoPasaje/ ListaTipos | Exp |
| PASAJE1 | nodoPasaje | Exp |

## 2.1 Gramática en el Algoritmo

Nuestra aproximación en la implementación de las reglas y atributos ha sido separar las estructuras de las reglas propias. Por ello, en el código, se pueden encontrar con dos paquetes tree y reglas. El primero contiene los nodos, listas y hojas necesarias que son utilizados como atributos, mientras que el segundo es el contender y aquel que hace uso de esas estructuras. Además, cada NoTerminal ha sido modelado como una clase, la cual implementa la interfaz ReglaSemantica y a través del método accionSemantica analiza en que producción se encuentra (en caso de tener más de una) para luego ejecutar la regla correspondiente.

Para la carga de los NoTerminales en la pila se utilizó java reflection y de esta manera se puede instanciar el objeto correspondiente solo con obtener el nombre de la clase en forma de String.

# III: Tabla de Símbolos

Para la tabla de símbolos se han implementado varios métodos. Cada uno para agregar una variable, constante, función, etc a un contexto en particular, al global. Buscar variables…

La estructura de la tabla consta de 3 Tablas. La tabla global y la tabla de Funciones y procedimientos que son de tipo TablaID y un HashMap de TablaID cuya clave es el contexto de tipo String. Se ha decidido abrir la tabla en global y funciones para así poder llamar una variable del mismo modo que una función.

A su vez, cada TablaID es un HashMap que contiene un objeto representativo de una variable, constante, parámetro, función o procedimiento y cuya clave es un String representando el ID de ese objeto.

## 3.1 Manejo de Recursividad

Se ha tenido que realizar un tweak para poder representar la recursividad, no decimos que estamos orgullosos de la barbaridad que se hizo pero, como el Bloque de sentencias se encuentra por debajo de la declaración, es decir que una sentencia A := proc(); se encuentra dentro de la definición de Proc(), se produce la ejecución de la regla semántica de la asignación antes que la declaración de la función y por ende no se encuentra en la tabla. Por esta razón, se ha añadido a la tabla unos campos Recur los cuales almacenan el ID, el tipo de retorno y los parámetros de esa función o procedimiento para poder realizar el chequeo de tipo de forma anticipada, permitiendo así el uso de Recursividad en el lenguaje.

## 3.2 Manejo del Contexto

Para el manejo del contexto se ha inctroducido una variable similar a una variable de Estado en la tabla de símbolos, y una agregar o buscar una variable, contante, etc lo hara primero en ese contexto y luego en el global, en el caso de agregarla la agregara solo en ese contexto (salvo Función, Procedimiento o Array).

# IV: Proceso De Implementación

Ya se ha comentado la mayoría del proceso de implementación y diseño. La ejecución de estas reglas se introdujo en la clase PilaLR, ya que la misma realiza las reducciones, momento en el cual se ejecuta la regla semántica.

## 4.1 Mecanismo De Recuperación De Errores

Para el manejo de errores semánticos se han implementado las excepciones TypeException y SemanticException. Una vez lanzadas las mismas se corta la ejecución de la regla y se captura propiamente el error para informarle al usuario por pantalla y luego seguir con el análisis del resto del código. Al encontrar un error de Tipo solucionable, además de lanzar la excepción se arregla ese error de tipos para que no se encadenen los errores semántico.

Para el manejo de errores algorítmicos se ha implementado la clase AlgorithmicError que más que nada es una clase de precaución y debug.

Para el manejo de errores en la tabla de símbolos se ha implementado la clase TableException que dependiendo el tipo de error es capturada y transformado en un Semantic o Type Exception o un AlgorithmicException.

# V: Pruebas

Las pruebas realizadas se encuentras anexas en la carpeta Test.

# VI: Cambios Realizados al Sintáctico

Se han realizado cambios en la gramática más que nada. Los cambios realizados han sido aquyos sugeridos por el profesor en la corrección de la entrega pasada y además hemos detectado un error en la producción Pasaje el cual ha sido debidamente corregido.

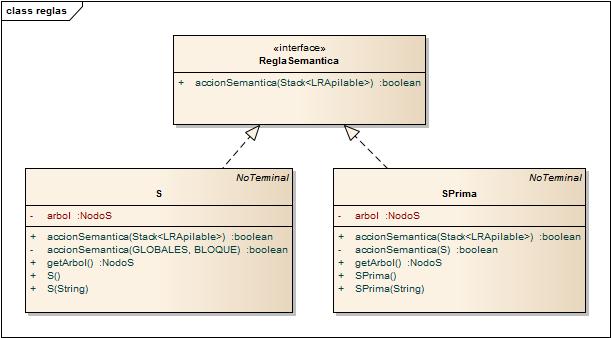
Además, se modificó la salida del árbol para que en la carpeta output ahora se encuentre el árbol de ejecución y no el árbol sintáctico.

# VII: Conclusiones

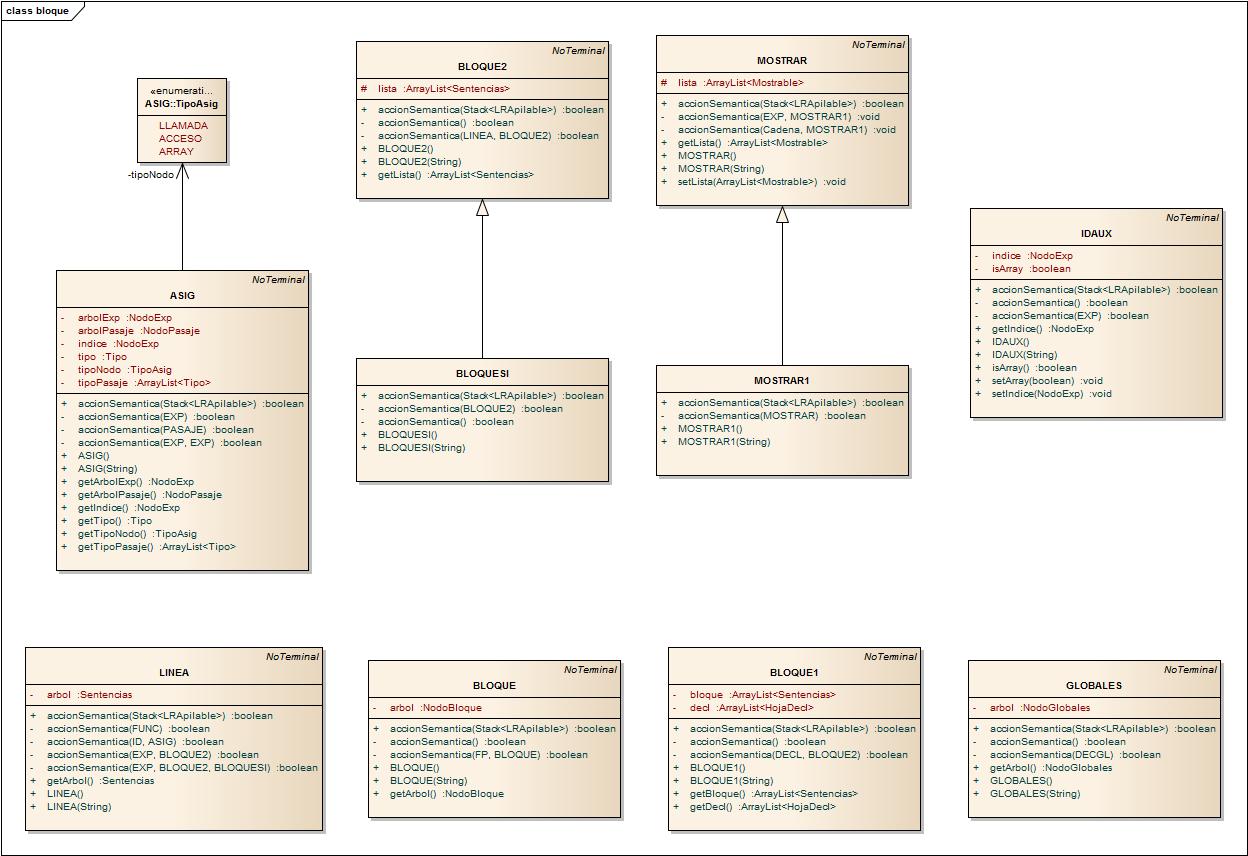
El problema principal que se ha tenido ha sido el modelado de las clases ya que las mismas no solo eran una gran cantidad (80) sino que las mismas estaban altamente relacionados, la estructura con los No Terminales. Si bien se cree que el diseño puede ser mejorado, esta solución brinda alta flexibilidad y facilidad de modificación.

# A: Anexo Diagramas de Clases

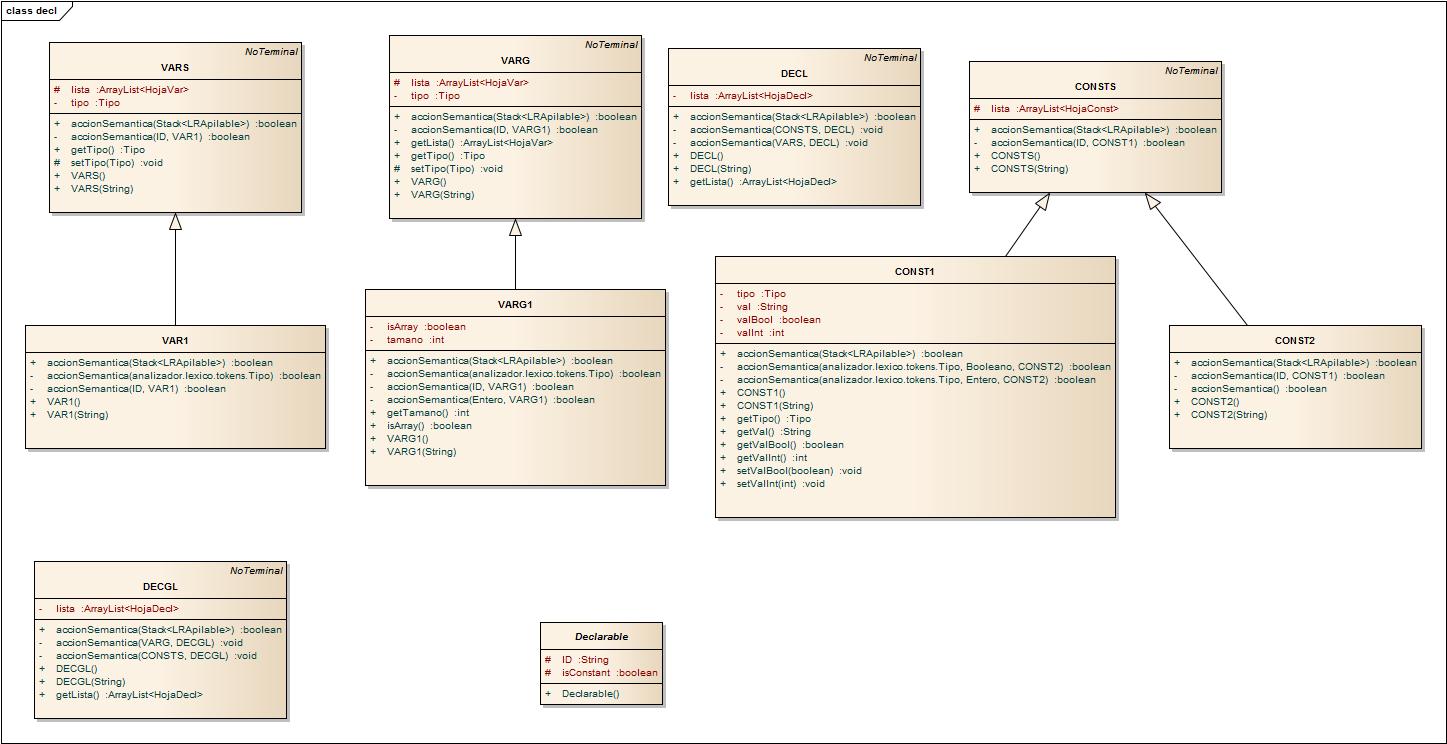
## A.1: Reglas y No Terminales



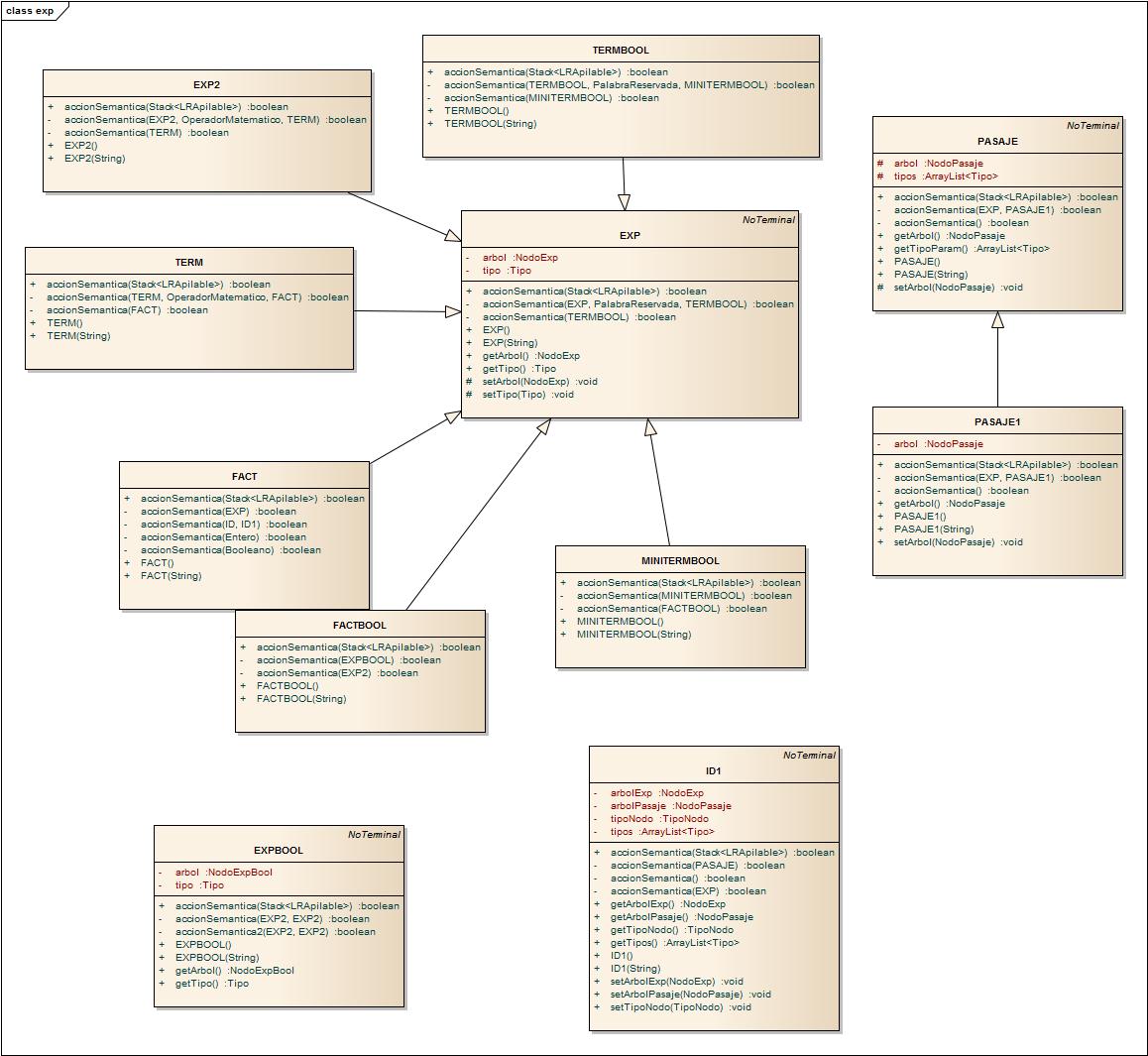
### A.1.1: Bloque



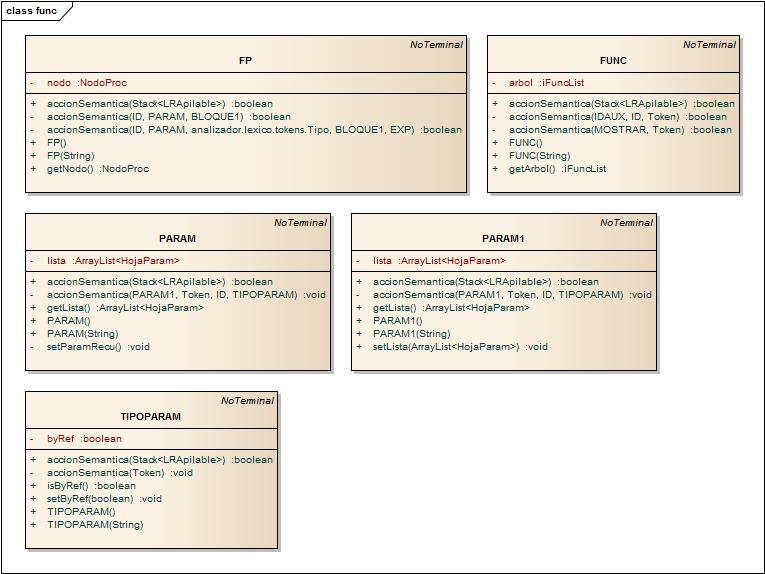
### A.1.2: Decl



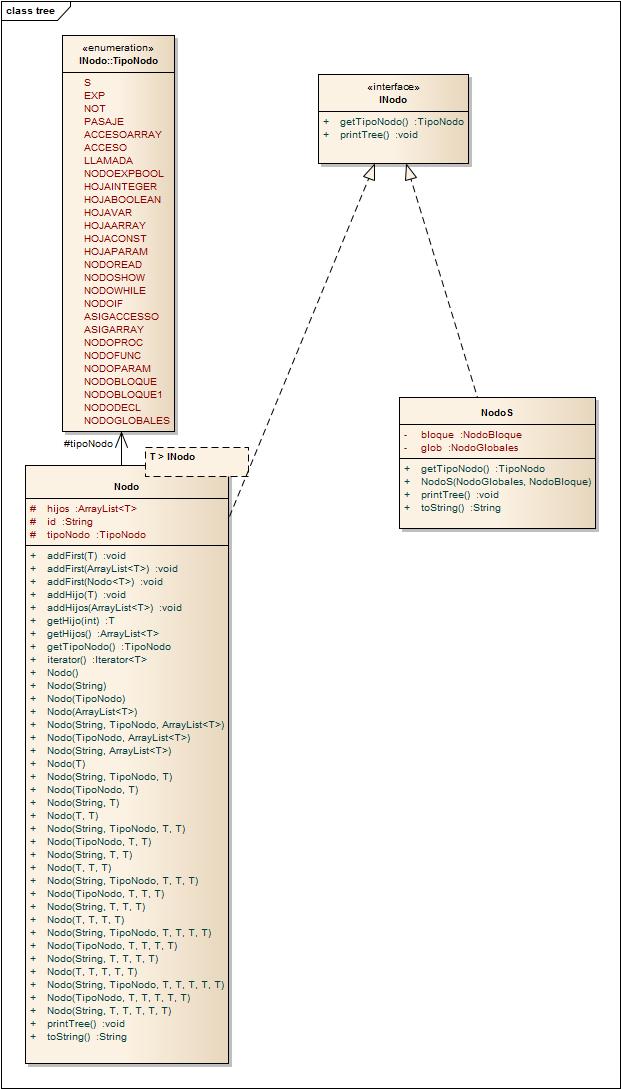
### A.1.3 Exp



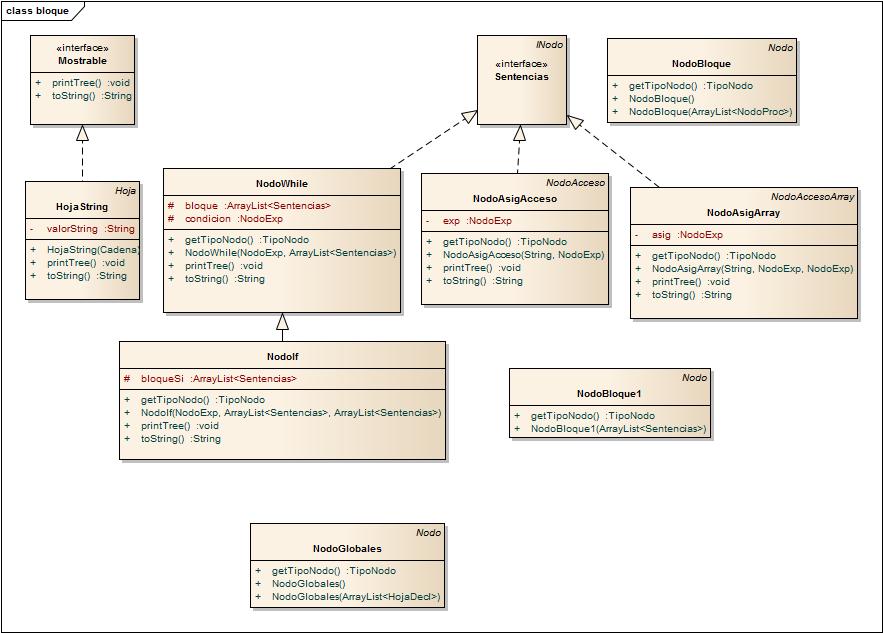
### A.1.4: Func



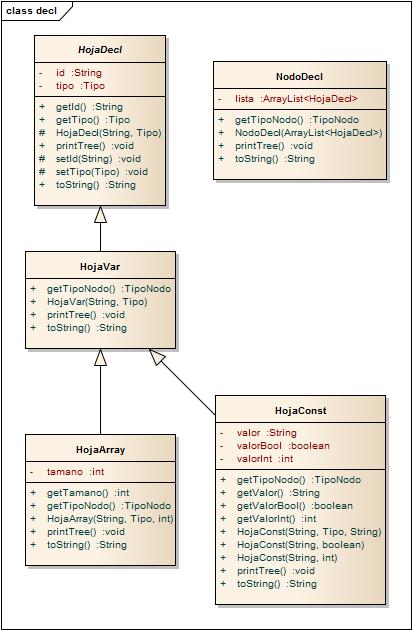
## A.2: Nodos y Estructuras



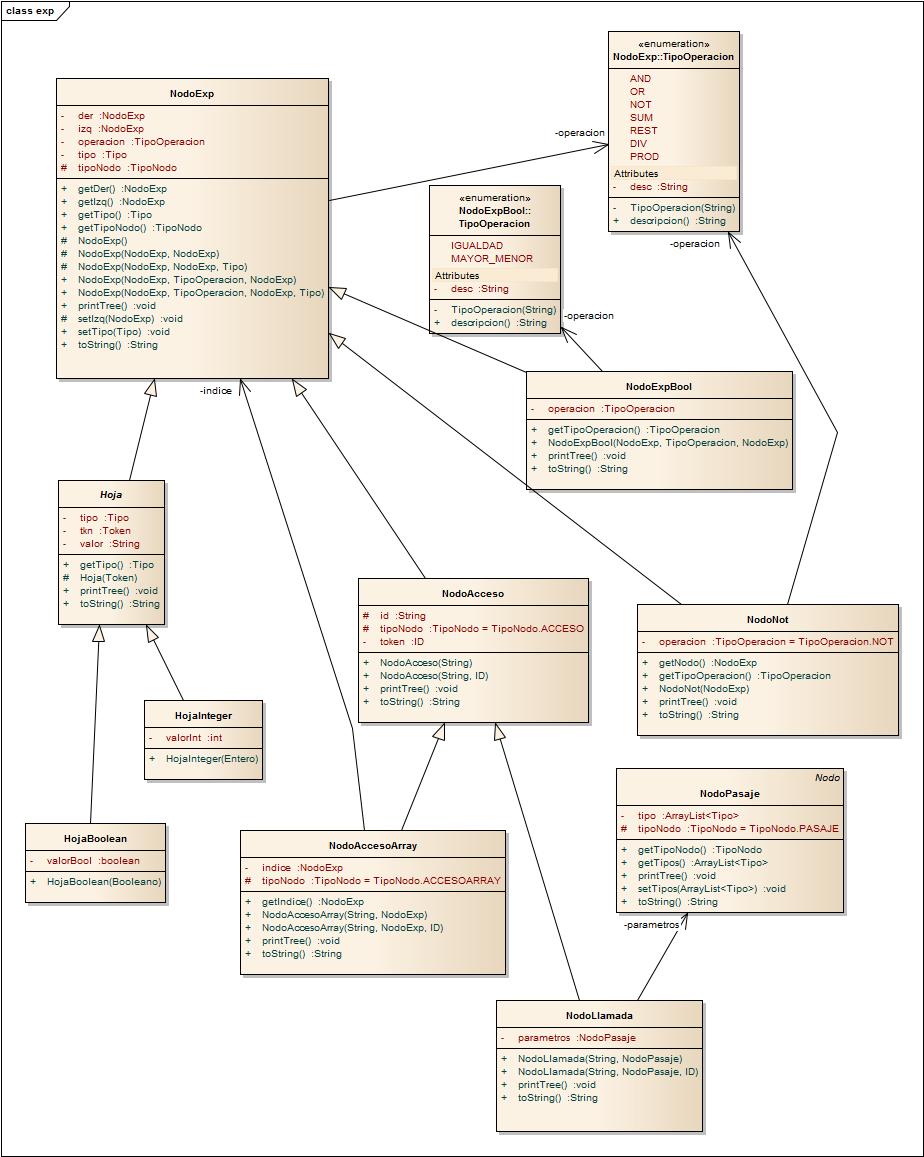
### A.2.1 Bloque



### A.2.3 Decl



### A.2.3 EXP



### A.2.4: FUNC

