Compilador MiniJava:

Manual técnico

Compiladores e Interpretes 2018

Rodrigo Díaz Figueroa

LU 97400

Indice

Compilación

Para compilar el programa desde consola se deberá ejecutar el siguiente comando desde la ubicación de los archivos fuente

```
>javac Minijavac.java
```

Ejecución

Para la ejecución del compilador Minijava desde consola se deberá ejecutar el siguiente comando

```
>java Minijavac <archivo fuente> [<archivo salida>]
```

Donde **<archivo fuente>** es el path absoluto o relativo dependiendo de donde se encuentra el archivo fuente a analizar en el equipo.

Y el opcional, **<archivo salida>** es el nombre del archivo de salida donde se escribirá la salida del programa.

Alfabeto de entrada

El alfabeto reconocido por el analizador léxico como parte del lenguaje son los símbolos del código ASCII Extendido imprimibles.

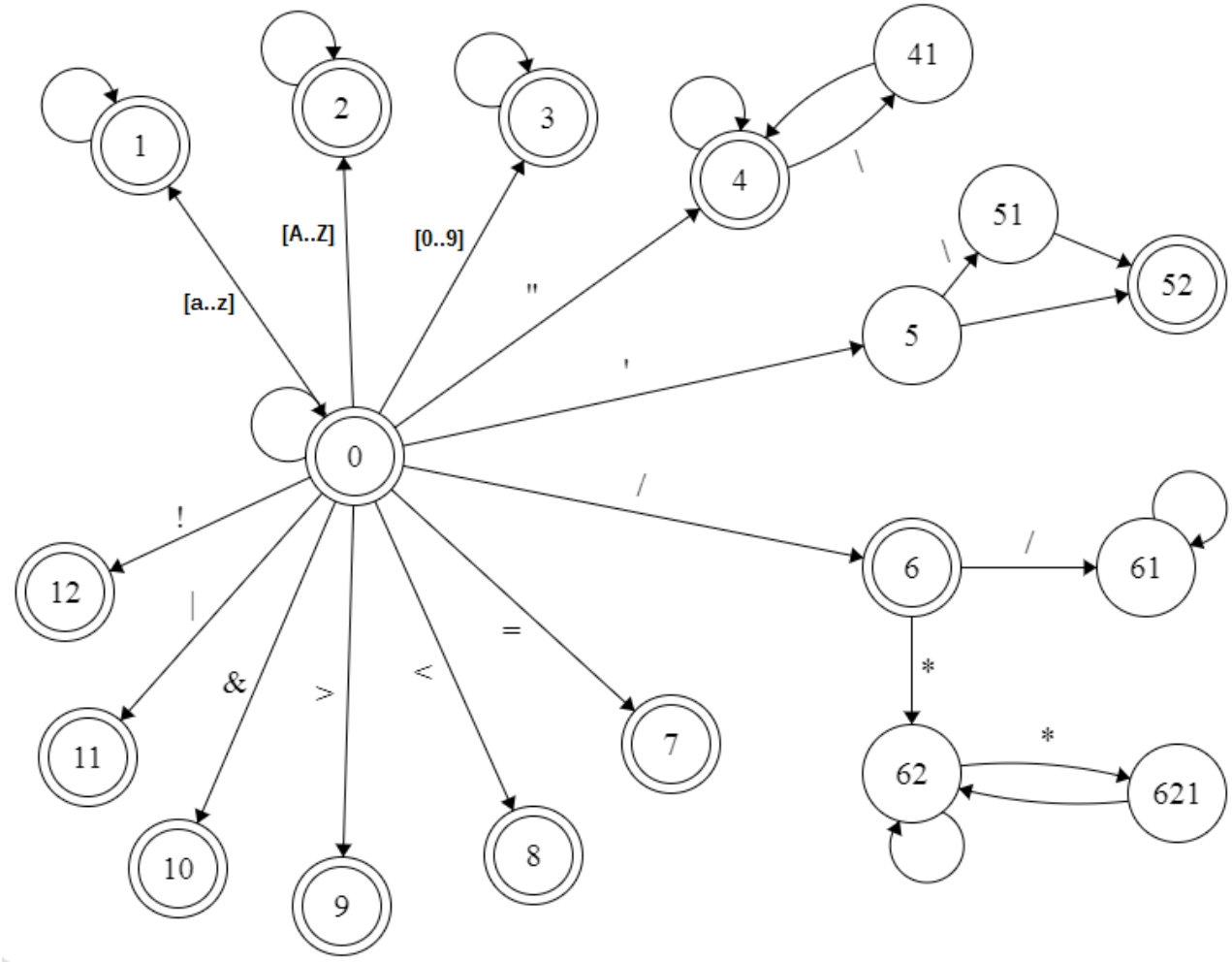
Tokens reconocidos

Nombre	Expresión regular
TT_IDClase	[A..Z][Letras + Dígitos + _]*
TT_IDMetVar	[a..z][Letras + Dígitos + _]*
TT_LitEntero	[0..9][0..9]*
TT_LitString	“[Caracter(1)]*”
TT_LitCaracter	‘[Caracter(2)]’
TT_LitCaracter	‘\[Caracter(1)]’
TT_PunPuntoComa	;
TT_PunPunto	-
TT_PunComa	,
TT_PunLlave_A	{
TT_PunLlave_C	}
TT_PunParent_A	(
TT_PunParent_C	)
TT_PunCorch_A	[
TT_PunCorch_C	]
TT_AsigIgual	=
TT_OpMayor	>
TT_OpMenor	<
TT_OpMayorIg	>=
TT_OpMenorIg	<=
TT_OpDesigual	!=
TT_OpDobleIgual	==
TT_OpNegBool	!
TT_OpSuma	+
TT_OpResta	-
TT_OpMult	*
TT_OpDiv	/
TT_OpAndDoble	&&

TT_OpOrDoble	||
TPC_Class	class
TPC_Extends	extends
TPC_Static	static
TPC_Dynamic	dynamic
TPC_String	String
TPC_Boolean	boolean
TPC_Char	char
TPC_Int	int
TPC_Public	public
TPC_Private	private
TPC_Void	void
TPC_Null	null
TPC_If	if
TPC_Else	else
TPC_While	while
TPC_Return	return
TPC_This	this
TPC_New	new
TPC_True	true
TPC_False	false

***Notas*:**
Dígitos: Cualquier número entre 0 y 9.
Letras: Cualquier letra mayúscula o minúscula.
Operador +: Indica la disyunción.
Caracter(1): Cualquier caracter.
Caracter(2): Cualquier caracter excepto la barra invertida, el salto de línea, o la comilla simple.

Autómata



Estados:
0: Reconocedor de la gran mayoría de los tokens de un solo carácter (Ej: + - * ; . , () { } [ ])
1: Reconocedor de tokens tipo Identificador Metodo-Variable y de todas los tokens de palabras clave excepto String
2: Reconocedor de tokens tipo Identificador Clase y del token palabra clave String
3: Reconocedor de tokens de tipo Literal Entero
4: Reconocedor de tokens de tipo Literal Cadena de caracteres
41: Ignora el carácter de escape en tokens de tipo Literal Cadena de caracteres
5: Pertenece a los estados reconocedores de tokens de tipo Literal Caracter
51: Ignora el carácter de escape en tokens de tipo Literal Caracter
52: Reconocedor de tokens de tipo Literal Caracter
6: Reconocedor de tokens de tipo Operador División
61: Ignora los comentarios simples
62: Pertenece a los estados que ignoran comentarios multilinea
621: Pertenece a los estados que ignoran comentarios multilinea
7: Reconocedor de tokens de tipo Asignación Igual y Operador Igual
8: Reconocedor de tokens de tipo Operador Menor y Menor Igual
9: Reconocedor de tokens de tipo Operador Mayor y Mayor Igual
10: Reconocedor del token de tipo Operador AND
11: Reconocedor del token de tipo Operador OR
12: Reconocedor de tokens de tipo Operador Desigual y Negación

Gramáticas

Gramática original:

```
<Inicial>::= <Clase>+
<Clase>::= class idClase <Herencia>? { <Miembro>* }
<Herencia>::= extends idClase
<Miembro>::= <Atributo> | <Ctor> | <Metodo>
<Atributo>::=<Visibilidad> <Tipo> <ListaDecVars> ;
<Metodo>::=<FormaMetodo> <TipoMetodo> idMetVar <ArgsFormales> <Bloque>
<Ctor>::= idClase <ArgsFormales> <Bloque>
<ArgsFormales>::= ( <ListaArgsFormales>? )
<ListaArgsFormales>::= <ArgFormal>
<ListaArgsFormales>::= <ArgFormal>, <ListaArgsFormales>
<ArgFormal>::= <Tipo> idMetVar
<FormaMetodo>::= static | dynamic
<Visibilidad>::= public | private
<TipoMetodo>::= <Tipo> | void
<Tipo>::=<TipoPrimitivo> | <TipoReferencia>
<TipoPrimitivo>::= boolean | char | int
<TipoReferencia>::= idClase | String | <TipoPrimitivo>[ ]
<ListaDecVars>::= idMetVar
<ListaDecVars>::= idMetVar , <ListaDecVars>
<Bloque>::= { <Sentencia>* }
<Sentencia>::= ;
<Sentencia>::= <Asignacion> ;
<Sentencia>::= <SentenciaLlamada> ;
<Sentencia>::= <Tipo> <ListaDecVars>;
<Sentencia>::= if ( <Expresion> ) <Sentencia>
<Sentencia>::= if ( <Expresion> ) <Sentencia> else <Sentencia>
<Sentencia>::= while ( <Expresion> ) <Sentencia>
<Sentencia>::= <Bloque>
<Sentencia>::= return <Expresion>? ;
<Asignacion>::= <AccesoVar> = <Expresion>
<Asignacion>::= <AccesoThis> = <Expresion>
<SentenciaLlamada>::= ( <Primario> )
<Expresion>::= <ExpOr>
<ExpOr>::= <ExpOr> || <ExpAnd> | <ExpAnd>
<ExpAnd>::= <ExpAnd> && <ExpIg> | <ExpIg>
<ExpIg>::=<ExpIg> <OpIg> <ExpComp> | <ExpComp>
<ExpComp>::=<ExpAd> <OpComp> <ExpAd> | <ExpAd>
<ExpAd>::= <ExpAd> <OpAd> <ExpMul> | <ExpMul>
<ExpMul>::= <ExpMul> <OpMul> <ExpUn> | <ExpUn>
<ExpUn>::= <OpUn> <ExpUn> | <Operando>
<OpIg>::= == | !=
<OpComp>::= < | > | <= | >=
<OpAd>::= + | -
<OpUn>::= + | - | !
<OpMul>::= * | /
<Operando>::= <Literal>
<Operando>::= <Primario>
<Literal>::= null | true | false | intLiteral | charLiteral |
stringLiteral
<Primario>::= <ExpresionParentizada>
```

```
<Primario>::= <AccesoThis>
<Primario>::= <AccesoVar>
<Primario>::= <LlamadaMetodo>
<Primario>::= <LlamadaMetodoEstatico>
<Primario>::= <LlamadaCtor>
<ExpresionParentizada>::= ( <Expresion> ) <Encadenado>?
<AccesoThis>::= this <Encadenado>?
<AccesoVar> ::= idMetVar <Encadenado>?
<LlamadaMetodo>::= idMetVar <ArgsActuales> <Encadenado>?
<LlamadaMetodoEstatico>::= idClase . <LlamadaMetodo> <Encadenado>?
<LlamdaCtor>::= new idClase <ArgsActuales> <Encadenado>?
<LlamdaCtor>::= new <TipoPrimitivo> [ <Expresion> ] <Encadenado>?
<ArgsActuales>::= ( <ListaExps>? )
<ListaExps>::= <Expresion>
<ListaExps>::= <Expresion> , <ListaExps>
<Encadenado>::= . <LlamadaMetodoEncadenado>
<Encadenado>::= . <AccesoVarEncadenado>
<Encadenado>::= <AccesoArregloEncadenado>
<LlamadaMetodoEncadenado>::= idMetVar <ArgsActuales> <Encadenado>?
<AccesoVarEncadenado>::= idMetVar <Encadenado>?
<AccesoArregloEncadenado>::= [ <Expresion> ] <Encadenado>?
```

Gramática sin BNF-extendido:

```
Inicio1::= Clase Inicio2
Inicio2::= Inicio1 | ε
Clase::= class idClase Herencia { Miembros }
Herencia::= extends idClase | ε
Miembros::= Miembro Miembros | ε
Miembro::= Atributo | Ctor | Metodo
Atributo::= Visibilidad Tipo ListaDecVars ;
Metodo::= FormaMetodo TipoMetodo idMetVar ArgsFormales Bloque
Ctor::= idClase ArgsFormales Bloque
ArgsFormales::= ( ListaArg )
ListaArg::= ε | Arg , ArgFormales
ArgFormales::= ε | Arg , ArgFormales
Arg::= Tipo idMetVar
FormaMetodo::= static | dynamic
Visibilidad::= public | private
TipoMetodo::= Tipo | void
Tipo::= TipoPrimitivo | TipoReferencia
TipoPrimitivo::= boolean | char | int
TipoReferencia::= idClase | String | TipoPrimitivo [ ]
ListaDecVars::= idMetVar
ListaDecVars::= idMetVar , ListaDecVars
Bloque::= { Sentencias }
Sentencias::= Sentencia Sentencias | ε
Sentencia::= ;
Sentencia::= Asignacion ;
Sentencia::= SentenciaLlamada ;
Sentencia::= Tipo ListaDecVars ;
Sentencia::= if ( Expresion ) Sentencia
Sentencia::= if ( Expresion ) Sentencia else Sentencia
Sentencia::= while ( Expresion ) Sentencia
Sentencia::= Bloque
```

```
Sentencia::= return Expresiones ;
Asignacion::= AccesoVar = Expresion
Asignacion::= AccesoThis = Expresion
SentenciaLlamada::= ( Primario )
Expresiones::= Expresion Expresiones | ε
Expresion::= ExpOr
ExpOr::= ExpOr || ExpAnd | ExpAnd
ExpAnd::= ExpAnd && ExpIg | ExpIg
ExpIg::= ExpIg OpIg ExpComp | ExpComp
ExpComp::= ExpAd OpComp ExpAd | ExpAd
ExpAd::= ExpAd OpAd ExpMul | ExpMul
ExpMul::= ExpMul OpMul ExpUn | ExpUn
ExpUn::= OpUn ExpUn | Operando
OpIg::= == | !=
OpComp::= < | > | <= | >=
OpAd::= + | -
OpUn::= + | - | !
OpMul::= * | /
Operando::= Literal
Operando::= Primario
Literal::= null | true | false | intLiteral | charLiteral | stringLiteral
Primario::= ExpresionParentizada
Primario::= AccesoThis
Primario::= AccesoVar
Primario::= LlamadaMetodo
Primario::= LlamadaMetodoEstatico
Primario::= LlamadaCtor
ExpresionParentizada::= ( Expresion ) Encadenado
AccesoThis::= this Encadenado
AccesoVar::= idMetVar Encadenado
LlamadaMetodo::= idMetVar ArgsActuales Encadenado
LlamadaMetodoEstatico::= idClase . LlamadaMetodo Encadenado
LlamdaCtor::= new idClase ArgsActuales Encadenado
LlamdaCtor::= new TipoPrimitivo [ Expresion ] Encadenado
ArgsActuales::= ( ListaExpresiones )
ListaExpresiones::= Expresion ListaExp | ε
ListaExp::= , Expresion ListaExp | ε
ListaExps::= Expresion
ListaExps::= Expresion , ListaExps
Encadenado::= ε | . Acceso | AccesoArregloEncadenado
Acceso::= LlamadaMetodoEncadenado | AccesoVarEncadenado
LlamadaMetodoEncadenado::= idMetVar ArgsActuales Encadenado
AccesoVarEncadenado::= idMetVar Encadenado
AccesoArregloEncadenado::= [ Expresion ] Encadenado
```

Gramática sin recursión a izquierda

```
Inicio1::= Clase Inicio2
Inicio2::= Inicio1 | ε
Clase::= class idClase Herencia { Miembros }
Herencia::= extends idClase | ε
Miembros::= Miembro Miembros | ε
Miembro::= Atributo | Ctor | Metodo
Atributo::= Visibilidad Tipo ListaDecVars ;
Metodo::= FormaMetodo TipoMetodo idMetVar ArgsFormales Bloque
```

```
Ctor::= idClase ArgsFormales Bloque
ArgsFormales::= ( ListaArg )
ListaArg::= ε | Arg , ArgFormales
ArgFormales::= ε | Arg , ArgFormales
Arg::= Tipo idMetVar
FormaMetodo::= static | dynamic
Visibilidad::= public | private
TipoMetodo::= Tipo | void
Tipo::= TipoPrimitivo | TipoReferencia
TipoPrimitivo::= boolean | char | int
TipoReferencia::= idClase | String | TipoPrimitivo [ ]
ListaDecVars::= idMetVar
ListaDecVars::= idMetVar , ListaDecVars
Bloque::= { Sentencias }
Sentencias::= Sentencia Sentencias | ε
Sentencia::= ;
Sentencia::= Asignacion ;
Sentencia::= SentenciaLlamada ;
Sentencia::= Tipo ListaDecVars ;
Sentencia::= if ( Expresion ) Sentencia
Sentencia::= if ( Expresion ) Sentencia else Sentencia
Sentencia::= while ( Expresion ) Sentencia
Sentencia::= Bloque
Sentencia::= return Expresiones ;
Asignacion::= AccesoVar = Expresion
Asignacion::= AccesoThis = Expresion
SentenciaLlamada::= ( Primario )
Expresiones::= Expresion Expresiones | ε
Expresion::= ExpOr
ExpOr::= ExpAnd ExpOrR
ExpOrR::= || ExpAnd ExpOrR | ε
ExpAnd::= ExpIg ExpAndR
ExpAndR::= && ExpIg ExpAndR | ε
ExpIg::= ExpComp ExpIgR
ExpIgR::= OpIgual ExpComp ExpIgR | ε
ExpComp::= ExpAd OpComp ExpAd | ExpAd
ExpAd::= ExpMul ExpAdR
ExpAdR::= OpAd ExpMul ExpAdR | ε
ExpMul::= ExpUn ExpMulR
ExpMulR::= OpMul ExpUn ExpMulR | ε
ExpUn::= OpUn ExpUn | Operando
OpIg::= == | !=
OpComp::= < | > | <= | >=
OpAd::= + | -
OpUn::= + | - | !
OpMul::= * | /
Operando::= Literal
Operando::= Primario
Literal::= null | true | false | intLiteral | charLiteral | stringLiteral
Primario::= ExpresionParentizada
Primario::= AccesoThis
Primario::= AccesoVar
Primario::= LlamadaMetodo
Primario::= LlamadaMetodoEstatico
Primario::= LlamadaCtor
ExpresionParentizada::= ( Expresion ) Encadenado
AccesoThis::= this Encadenado
AccesoVar::= idMetVar Encadenado
```

```
LlamadaMetodo::= idMetVar ArgsActuales Encadenado
LlamadaMetodoEstatico::= idClase . LlamadaMetodo Encadenado
LlamdaCtor::= new idClase ArgsActuales Encadenado
LlamdaCtor::= new TipoPrimitivo [ Expresion ] Encadenado
ArgsActuales::= ( ListaExpresiones )
ListaExpresiones::= Expresion ListaExp | ε
ListaExp::= , Expresion ListaExp | ε
ListaExps::= Expresion
ListaExps::= Expresion , ListaExps
Encadenado::= ε | . Acceso | AccesoArregloEncadenado
Acceso::= LlamadaMetodoEncadenado | AccesoVarEncadenado
LlamadaMetodoEncadenado::= idMetVar ArgsActuales Encadenado
AccesoVarEncadenado::= idMetVar Encadenado
AccesoArregloEncadenado::= [ Expresion ] Encadenado
```

Gramática sin prefijos comunes a izquierda

```
Inicio::= Clase Clases $
Clases::= Clase Clases
Clases::= ε
Clase::= class idClase Herencia { Miembros }
Herencia::= extends idClase
Herencia::= ε
Miembros::= Miembro Miembros
Miembros::= ε
Miembro::= Atributo
Miembro::= Ctor
Miembro::= Metodo
Atributo::= Visibilidad Tipo ListaDecVars Inicializacion ;
Inicializacion::= = Expresion
Inicializacion::= ε
Metodo::= FormaMetodo TipoMetodo idMetVar ArgsFormales Bloque
Ctor::= idClase ArgsFormales Bloque
ArgsFormales::= ( ListaArg )
ListaArg::= Arg ArgFormales
ListaArg::= ε
ArgFormales::= , Arg ArgFormales
ArgFormales::= ε
Arg::= Tipo idMetVar
FormaMetodo::= static
FormaMetodo::= dynamic
Visibilidad::= public
Visibilidad::= private
TipoMetodo::= Tipo
TipoMetodo::= void
Tipo::= boolean PosibleArreglo
Tipo::= char PosibleArreglo
Tipo::= int PosibleArreglo
Tipo::= idClase PosibleArreglo
Tipo::= String PosibleArreglo
PosibleArreglo::= [ ] PosibleArreglo /* Permitir matrices? */
PosibleArreglo::= ε
ListaDecVars::= idMetVar ListaDV
ListaDV::= , idMetVar ListaDV
```

```
ListaDV::= ε
Bloque::= { Sentencias }
Sentencias::= Sentencia Sentencias
Sentencias::= ε
Sentencia::= ;
Sentencia::= if ( Expresion ) Sentencia SentenciaElse
Sentencia::= while ( Expresion ) Sentencia
Sentencia::= return Expresiones ;
Sentencia::= Asignacion ;
Sentencia::= SentenciaLlamada ;
Sentencia::= Tipo ListaDecVars Inicializacion ;
Sentencia::= Bloque
SentenciaElse::= else Sentencia
SentenciaElse::= ε
Expresiones::= Expresion
Expresiones::= ε
Asignacion::= AccesoVar = Expresion
Asignacion::= AccesoThis = Expresion
SentenciaLlamada::= ( Primario )
Expresion::= ExpOr
ExpOr::= ExpAnd ExpOrR
ExpOrR::= || ExpAnd ExpOrR
ExpOrR::= ε
ExpAnd::= ExpIg ExpAndR
ExpAndR::= && ExpIg ExpAndR
ExpAndR::= ε
ExpIg::= ExpComp ExpIgR
ExpIgR::= OpIgual ExpComp ExpIgR
ExpIgR::= ε
ExpComp::= ExpAd ExpCompR
ExpCompR::= OpComp ExpAd
ExpCompR::= ε
ExpAd::= ExpMul ExpAdR
ExpAdR::= OpAd ExpMul ExpAdR
ExpAdR::= ε
ExpMul::= ExpUn ExpMulR
ExpMulR::= OpMul ExpUn ExpMulR
ExpMulR::= ε
ExpUn::= OpUn ExpUn
ExpUn::= Operando
OpIgual::= ==
OpIgual::= !=
OpComp::= <
OpComp::= >
OpComp::= <=
OpComp::= >=
OpAd::= +
OpAd::= -
OpUn::= +
OpUn::= -
OpUn::= !
OpMul::= *
OpMul::= /
Operando::= Literal
Operando::= Primario
Literal::= null
Literal::= true
Literal::= false
```

```
Literal::= intLiteral
Literal::= charLiteral
Literal::= stringLiteral
Primario::= idMetVar MetodoVariable
Primario::= ExpresionParentizada
Primario::= AccesoThis
Primario::= LlamadaMetodoEstatico
Primario::= LlamadaCtor
MetodoVariable::= ArgsActuales Encadenado
MetodoVariable::= Encadenado
ExpresionParentizada::= ( Expresion ) Encadenado
Encadenado::= . idMetVar Acceso
Encadenado::= AccesoArregloEncadenado
Encadenado::= ε
Acceso::= LlamadaMetodoEncadenado
Acceso::= AccesoVarEncadenado
AccesoThis::= this Encadenado
AccesoVar::= idMetVar Encadenado
LlamadaMetodo::= idMetVar ArgsActuales Encadenado
LlamadaMetodoEstatico::= idClase . LlamadaMetodo
LlamadaCtor::= new LlamadaCtorR
LlamadaCtorR::= idClase LlamadaCtorIDClase
LlamadaCtorR::= TipoArreglo [ Expresion ] Encadenado
LlamadaCtorIDClase::= ArgsActuales Encadenado
LlamadaCtorIDClase::= [ Expresion ] Encadenado
TipoArreglo::= char
TipoArreglo::= boolean
TipoArreglo::= int
TipoArreglo::= String
ArgsActuales::= ( ListaExpresiones )
ListaExpresiones::= Expresion ListaExp
ListaExpresiones::= ε
ListaExp::= , Expresion ListaExp
ListaExp::= ε
LlamadaMetodoEncadenado::= ArgsActuales Encadenado
AccesoVarEncadenado::= Encadenado
AccesoArregloEncadenado::= [ Expresion ] Encadenado
```

Esquema de Traducción (EDT)

Chequeo de declaraciones

{Antes de inicio se crean las clases predefinidas Object y System con sus respectivos metodos y se agregan a la Tabla de Simbolos}
Inicio → Clase Clases $
{Luego de inicio se consolida la Tabla de simbolos}

Clase → class idClase {Se guarda el token con el nombre de la clase} **Herencia**
{Se guarda el token con el nombre del padre de la clase que retorna Herencia, se crea la clase, se agrega a la Tabla de Simbolos y se la setea como clase actual}
{ Miembros }

Herencia → extends {Se guarda el token de nombre de la clase} idClase {Herencia retorna el token de nombre de clase}
Herencia → ε {Se retorna un token nulo}

Atributo → Visibilidad {Se guarda la visibilidad} **Tipo** {Se guarda el tipo y se crea una lista de variables con el tipo para pasarle a ListaDecVars}
ListaDecVars {Se crean los atributos con la lista de nombres de variables y tipo y se agrega a la clase actual} **Inicializacion ;**

Metodo → FormaMetodo {Se guarda la forma del metodo} **TipoMetodo** {Se guarda el tipo de retorno del metodo}
idMetVar {Se guarda el token del nombre del metodo, se crea un metodo con forma, tipo de retorno y nombre y se lo setea como miembro actual} **ArgsFormales**
{Se agrega el metodo a la tabla de simbolos} **Bloque**

Ctor → idClase {Se guarda el token del nombre del constructor, se crea el constructor y se lo setea como miembro actual}
ArgsFormales {Se agrega el constructor a la tabla de simbolos} **Bloque**

Arg → Tipo {Se guarda el tipo del parametro} **idMetVar** {Se guarda el token del nombre del parametro, se crea el parametro y se agrega al miembro actual}

FormaMetodo → static {Retorna forma static}

FormaMetodo → dynamic {Retorna forma dynamic}

Visibilidad → public {Retorna visibilidad public}

Visibilidad → private {Retorna visibilidad private}

TipoMetodo → Tipo {Retorna el tipo que retorna Tipo}

TipoMetodo → void {Se crea un tipo simple con el token de nombre void y se retorna}

Tipo → boolean {Se crea un tipo simple con el token de nombre boolean y se pasa como parametro a PosibleArreglo}
PosibleArreglo {Retorna el tipo que retorna PosibleArreglo}

Tipo → **char** Se crea un tipo simple con el token de nombre char y se pasa como parametro a PosibleArreglo}
PosibleArreglo {Retorna el tipo que retorna PosibleArreglo}

Tipo → **int** {Se crea un tipo simple con el token de nombre int y se pasa como parametro a PosibleArreglo}
PosibleArreglo {Retorna el tipo que retorna PosibleArreglo}

Tipo → **idClase** {Se crea un tipo clase con el token de nombre de la clase y se pasa como parametro a PosibleArreglo}
PosibleArreglo {Retorna el tipo que retorna PosibleArreglo}

Tipo → **String** {Se crea un tipo simple con el token de nombre String y se pasa como parametro a PosibleArreglo}
PosibleArreglo {Retorna el tipo que retorna PosibleArreglo}

PosibleArreglo → **[ ]** {Setea que es arreglo al tipo recibido como parametro y se retorna}

PosibleArreglo → **ε** {Retorna el tipo recibido como parametro}

{Recibe como parametros un tipo y una lista de parametros}
ListaDecVars → **idMetVar** {Se guarda el token de nombre del parametro, se crea un parametro con ese nombre y el tipo y se agrega a la lista}
{Se pasa como parametro el tipo y la lista a ListaDV} **ListaDV**

{Recibe como parametros un tipo y una lista de parametros}
ListaDV → **, idMetVar** {Se guarda el token de nombre del parametro, se crea un parametro con ese nombre y el tipo y se agrega a la lista}
{Se pasa como parametro el tipo y la lista a ListaDV} **ListaDV**

Chequeo de sentencias

Atributo →
Visibilidad Tipo ListaDecVars
Inicializacion
{Se guarda el NodoExpresion que retorna inicializacion}
{Si el NodoExpresion no es nulo se setea que es un valor de atributo}
{Se crean los atributos en la clase actual con el NodoExpresion como valor}
;

Inicializacion → **= Expresion** {Se retorna el NodoExpresion que retorna Expresion}
Inicializacion → **ε** {Se retorna null}

Metodo →
FormaMetodo TipoMetodo idMetVar ArgsFormales
{Antes de llamar a bloque se setea al bloque actual como nulo}
Bloque {Se guarda el NodoBloque que retorna Bloque}
{Se setea el NodoBloque guardado al miembro actual}

Ctor →
idClase ArgsFormales
{Antes de llamar a bloque se setea al bloque actual como nulo}
Bloque {Se guarda el NodoBloque que retorna Bloque}

{Se setea el NodoBloque guardado al miembro actual}

Bloque →
{
{Se crea un nuevo NodoBloque con el bloque actual como padre}
{Se setea al NodoBloque recien creado como bloque actual}
Sentencias
}
{Se setea al padre del bloque actual, como bloque actual}
{Se retorna el NodoBloque creado}

Sentencias →
Sentencia {Se guarda el NodoSentencia que retorna Sentencia}
{Se agrega el NodoSentencia al bloque actual}
Sentencias

Sentencias → **ε**

Sentencia →
; {Se crea y se retorna un NodoPuntoComa}

Sentencia →
if ({Se crea un nuevo NodoIf con el token "if"}
Expresion {Se setea el NodoExpresion que retorna Expresion como condicion del NodoIf}
) Sentencia {Se setea el NodoSentencia que retorna Sentencia como sentencia then del NodoIf}
SentenciaElse {Se setea el NodoSentencia que retorna Sentencia como sentencia else del NodoIf}
{Se retorna el NodoIf creado}

Sentencia →
while ({Se crea un nuevo NodoWhile con el token "while"}
Expresion {Se setea el NodoExpresion que retorna Expresion como condicion del NodoWhile}
) Sentencia {Se setea el NodoSentencia que retorna Sentencia como sentencia del NodoWhile}
{Se retorna el NodoWhile creado}

Sentencia →
return {Se crea un nuevo NodoReturn con el token "return"}
Expresiones {Se setea el NodoExpresion que retorna Expresiones como expresion del NodoReturn}
;
{Se retorna el NodoReturn creado}

Sentencia →
Asignacion {Se guarda el NodoAsignacion que retorna Asignacion}
;
{Se retorna el nodo guardado}

Sentencia →
{Se crea un nuevo NodoSentenciaSimple}
SentenciaLlamada {Se setea el NodoExpresion que retorna SentenciaLlamada como expresion del NodoSentenciaSimple}
;
{Se retorna el NodoSentenciaSimple creado}

Sentencia →
Tipo {Se guarda el Tipo que retorna}

{Se crea una nueva lista de EParametro para pasarle a ListaDecVars}
{Se llama a ListaDecVars con la lista, y el tipo guardado} **ListaDecVars**
{Se crea un nuevo NodoDecVars con el Tipo y la lista}
Inicializacion {Se setea el NodoExpresion que retorna Inicializacion como valor del NodoDecVars} **;**
{Se retorna el NodoDecVars creado}

Sentencia → **Bloque** {Se retorna el NodoBloque que retorna Bloque}

SentenciaElse → **else Sentencia** {Se retorna el NodoSentencia que retorna Sentencia}
SentenciaElse → **ε** {Se retorna null}

Expresiones → **Expresion** {Se retorna el NodoExpresion que retorna Expresion}
Expresiones → **ε** {Se retorna null}

Asignacion →
{Se crea un nuevo NodoAsignacion}
AccesoVar {Se guarda el NodoVar que retorna}
{Se setea que el NodoVar esta del lado izquierdo}
{Se setea el NodoVar como lado izquierdo del NodoAsignacion}
= {Se setea el token "=" en el NodoAsignacion}
Expresion {Se setea el NodoExpresion que retorna Expresion como lado derecho del NodoAsignacion}
{Se retorna el NodoAsignacion creado}

Asignacion →
{Se crea un nuevo NodoAsignacion}
AccesoThis {Se guarda el NodoThis que retorna}
{Se setea que el NodoThis esta del lado izquierdo}
{Se setea el NodoThis como lado izquierdo del NodoAsignacion}
= {Se setea el token "=" en el NodoAsignacion}
Expresion {Se setea el NodoExpresion que retorna Expresion como lado derecho del NodoAsignacion}
{Se retorna el NodoAsignacion creado}

SentenciaLlamada →
(Primario {Se guarda el NodoExpresion que retorna Primario}
) {Se retorna el NodoExpresion guardado}

Expresion → **ExpOr** {Se retorna el NodoExpresion que retorna ExpOr}

ExpOr →
ExpAnd {Se guarda el NodoExpresion que retorna}
{Se llama a ExpOrR con el nodo guardado} **ExpOrR**
{Se retorna el NodoExpresion que retorna ExpOrR}

ExpOrR →
|| {Se crea un nuevo OpBool con el token}
{Se crea un nuevo NodoExpBin con el OpBool y el NodoExpresion recibido como parametro como lado izquierdo}
ExpAnd {Se setea el NodoExpresion que retorna como lado derecho del NodoExpBin}
{Se llama a ExpOrR con el NodoExpBin creado} **ExpOrR**
{Se retorna el NodoExpresion que retorna ExpOrR}

ExpOrR → **ε** {Se retorna el NodoExpresion recibido como parametro}

ExpAnd →
ExpIg {Se guarda el NodoExpresion que retorna}
{Se llama a ExpAndR con el nodo guardado} **ExpAndR**
{Se retorna el NodoExpresion que retorna ExpAndR}

ExpAndR →
&& {Se crea un nuevo OpBool con el token}
{Se crea un nuevo NodoExpBin con el OpBool y el NodoExpresion recibido como parametro como lado izquierdo}
ExpIg {Se setea el NodoExpresion que retorna como lado derecho del NodoExpBin}
{Se llama a ExpAndR con el NodoExpBin creado} **ExpAndR**
{Se retorna el NodoExpresion que retorna ExpAndR}

ExpAndR → **ε** {Se retorna el NodoExpresion recibido como parametro}

ExpIg →
ExpComp {Se guarda el NodoExpresion que retorna}
{Se llama a ExpIgR con el nodo guardado} **ExpIgR**
{Se retorna el NodoExpresion que retorna ExpIgR}

ExpIgR →
OpIgual {Se guarda el Operador que retorna}
{Se crea un nuevo NodoExpBin con el Operador y el NodoExpresion recibido como parametro como lado izquierdo}
ExpComp {Se setea el NodoExpresion que retorna como lado derecho del NodoExpBin}
{Se llama a ExpIgR con el NodoExpBin creado} **ExpIgR**
{Se retorna el NodoExpresion que retorna ExpIgR}

ExpIgR → **ε** {Se retorna el NodoExpresion recibido como parametro}

ExpComp →
ExpAd {Se guarda el NodoExpresion que retorna}
{Se llama a ExpCompR con el nodo guardado} **ExpCompR**
{Se retorna el NodoExpresion que retorna ExpCompR}

ExpCompR →
OpComp {Se guarda el Operador que retorna}
{Se crea un nuevo NodoExpBin con el Operador y el NodoExpresion recibido como parametro como lado izquierdo}
ExpAd {Se setea el NodoExpresion que retorna como lado derecho del NodoExpBin}
{Se retorna el NodoExpBin creado}

ExpCompR → **ε** {Se retorna el NodoExpresion recibido como parametro}

ExpAd →
ExpMul {Se guarda el NodoExpresion que retorna}
{Se llama a ExpAdR con el nodo guardado} **ExpAdR**
{Se retorna el NodoExpresion que retorna ExpAdR}

ExpAdR →
OpAd {Se guarda el Operador que retorna}
{Se crea un nuevo NodoExpBin con el Operador y el NodoExpresion recibido como parametro como lado izquierdo}
ExpMul {Se setea el NodoExpresion que retorna como lado derecho del NodoExpBin}
{Se llama a ExpAdR con el NodoExpBin creado} **ExpAdR**

{Se retorna el NodoExpresion que retorna ExpAdR}

ExpAdR → **ε** {Se retorna el NodoExpresion recibido como parametro}

ExpMul →
ExpUn {Se guarda el NodoExpresion que retorna}
{Se llama a ExpMulR con el nodo guardado} **ExpMulR**
{Se retorna el NodoExpresion que retorna ExpMulR}

ExpMulR →
OpMul {Se guarda el Operador que retorna}
{Se crea un nuevo NodoExpBin con el Operador y el NodoExpresion recibido como parametro como lado izquierdo}
ExpUn {Se setea el NodoExpresion que retorna como lado derecho del NodoExpBin}
{Se llama a ExpMulR con el NodoExpBin creado} **ExpMulR**
{Se retorna el NodoExpresion que retorna ExpMulR}

ExpMulR → **ε** {Se retorna el NodoExpresion recibido como parametro}

ExpUn → **OpUn ExpUn**
OpUn {Se guarda el Operador que retorna}
{Se crea un nuevo NodoExpUn con el Operador}
ExpUn {Se setea el NodoExpresion que retorna como expresion del NodoExpBin}
{Se retorna el NodoExpUn creado}

ExpUn → **Operando** {Se retorna el NodoExpresion que retorna Operando}

OpIgual → **== | !=** {Se crea y se retorna un OpComp con el token}

OpComp → **< | > | <= | >=** {Se crea y se retorna un OpCompMat con el token}

OpAd → **+ | -** {Se crea y se retorna un OpMat con el token}

OpUn → **+ | -** {Se crea y se retorna un OpMat con el token}

OpUn → **!** {Se crea y se retorna un OpBool con el token}

OpMul → *** | /** {Se crea y se retorna un OpMat con el token}

Operando → **Literal** {Se retorna el NodoExpresion que retorna Literal}
Operando → **Primario** {Se retorna el NodoExpresion que retorna Primario}

Literal → **null** {Se crea y se retorna un NodoLitNull con el token}
Literal → **true** {Se crea y se retorna un NodoLitBool con el token}
Literal → **false** {Se crea y se retorna un NodoLitBool con el token}
Literal → **intLiteral** {Se crea y se retorna un NodoLitInt con el token}
Literal → **charLiteral** {Se crea y se retorna un NodoLitChar con el token}
Literal → **stringLiteral** {Se crea y se retorna un NodoLitString con el token}

Primario →
idMetVar {Se guarda el token}
{Se llama a MetodoVariable con el token} **MetodoVariable**
{Se retorna el NodoPrimario que retorna MetodoVariable}

Primario → **ExpresionParentizada** {Se retorna el NodoPrimario que retorna ExpresionParentizada}

Primario → **AccesoThis** {Se retorna el NodoPrimario que retorna AccesoThis}

Primario → **LlamadaMetodoEstatico** {Se retorna el NodoPrimario que retorna LlamadaMetodoEstatico}

Primario → **LlamadaCtor** {Se retorna el NodoPrimario que retorna LlamadaCtor}

MetodoVariable →
{Se crea un nuevo NodoLlamadaDirecta con el token recibido como parametro}
{Se llama a ArgsActuales con la lista de argumentos del nodo creado}
ArgsActuales
Encadenado {Se setea el Encadenado que retorna como encadenado del NodoLlamadaDirecta}
{Se retorna el NodoLlamadaDirecta creado}

MetodoVariable →
{Se crea un nuevo NodoVar con el token recibido como parametro}
Encadenado {Se setea el Encadenado que retorna como encadenado del NodoVar}
{Se retorna el NodoVar creado}

ExpresionParentizada →
{Se crea un nuevo NodoExpParent}
(Expresion {Se setea al NodoExpresion retornado como expresion del NodoExpParent}
) Encadenado {Se setea al Encadenado retornado como encadenado del NodoExpParent}
{Se retorna el nodo creado}

Encadenado →
. idMetVar {Se guarda el token idMetVar}
{Se llama a Acceso con el token guardado}
Acceso {Se retorna el Encadenado retornado por Acceso}

Encadenado →**AccesoArregloEncadenado** {Se retorna el Encadenado retornado por AccesoArregloEncadenado}

Encadenado → **ε** {Se retorna null}

Acceso →
{Se llama a LlamadaMetodoEncadenado con el token recibido como parametro}
LlamadaMetodoEncadenado {Se retorna el Encadenado retornado por LlamadaMetodoEncadenado}

Acceso →
{Se llama a AccesoVarEncadenado con el token recibido como parametro}
AccesoVarEncadenado {Se retorna el Encadenado retornado por AccesoVarEncadenado}

AccesoThis →
this {Se guarda el token}
{Se crea un nuevo NodoThis con el token y la clase actual}
Encadenado {Se setea el Encadenado retornado como encadenado del NodoThis}
{Se retorna el NodoThis creado}

AccesoVar →
idMetVar {Se guarda el token}
{Se crea un nuevo NodoVar con el token}

Encadenado {Se setea el Encadenado retornado como encadenado del NodoVar}
{Se retorna el NodoVar creado}

LlamadaMetodoEstatico →
idClase {Se guarda el token}
. {Se llama a LlamadaMetodo con el token guardado]
LlamadaMetodo {Se retorna el NodoLlamEstat que retorna LlamadaMetodo}

LlamadaMetodo →
idMetVar {Se guarda el token}
{Se crea un nuevo NodoLlamEstat con el token guardado y el recibido como parametro}
{Se llama a ArgsActuales con la lista de argumentos del nodo recien creado}
ArgsActuales
Encadenado {Se setea el Encadenado retornado como encadenado del NodoLlamEstat}
{Se retorna el NodoLlamEstat creado}

LlamadaCtor → **new LlamadaCtorR** {Se retorna el NodoConst que retorna LlamadaCtorR}

LlamadaCtorR →
idClase {Se guarda el token}
{Se llama a LlamadaCtorIDClase con el token}
LlamadaCtorIDClase {Se retorna el NodoConst que retorna LlamadaCtorIDClase}

LlamadaCtorR →
{Se crea un nuevo NodoConstArray con el token actual}
TipoArreglo
[Expresion {Se setea al NodoExpresion retornado como tamaño del NodoConstArray}
] Encadenado {Se setea al Encadenado retornado como encadenado del NodoConstArray}
{Se retorna el NodoConstArray creado}

LlamadaCtorIDClase →
{Se crea un nuevo NodoConstArray con el token recibido como parametro}
[Expresion {Se setea al NodoExpresion retornado como tamaño del NodoConstArray}
] Encadenado {Se setea al Encadenado retornado como encadenado del NodoConstArray}
{Se retorna el NodoConstArray creado}

LlamadaCtorIDClase →
{Se crea un nuevo NodoConstComun con el token recibido como parametro}
{Se llama a ArgsActuales con la lista de argumentos vacia del nodo recien creado}
ArgsActuales
Encadenado {Se setea al Encadenado retornado como encadenado del NodoConstComun}
{Se retorna el NodoConstComun creado}

LlamadaMetodoEncadenado →
{Se crea un nuevo NodoLlamEncad con el token recibido como parametro}
{Se llama a ArgsActuales con la lista de argumentos vacia del nodo recien creado}
ArgsActuales
Encadenado {Se setea al Encadenado retornado como encadenado del NodoLlamEncad}
{Se retorna el NodoLlamEncad creado}

AccesoVarEncadenado →
{Se crea un nuevo NodoVarEncad con el token recibido como parametro}
Encadenado {Se setea al Encadenado retornado como encadenado del NodoVarEncad}
{Se retorna el NodoVarEncad creado}

AccesoArregloEncadenado →

{Se crea un nuevo NodoArregloEncad}

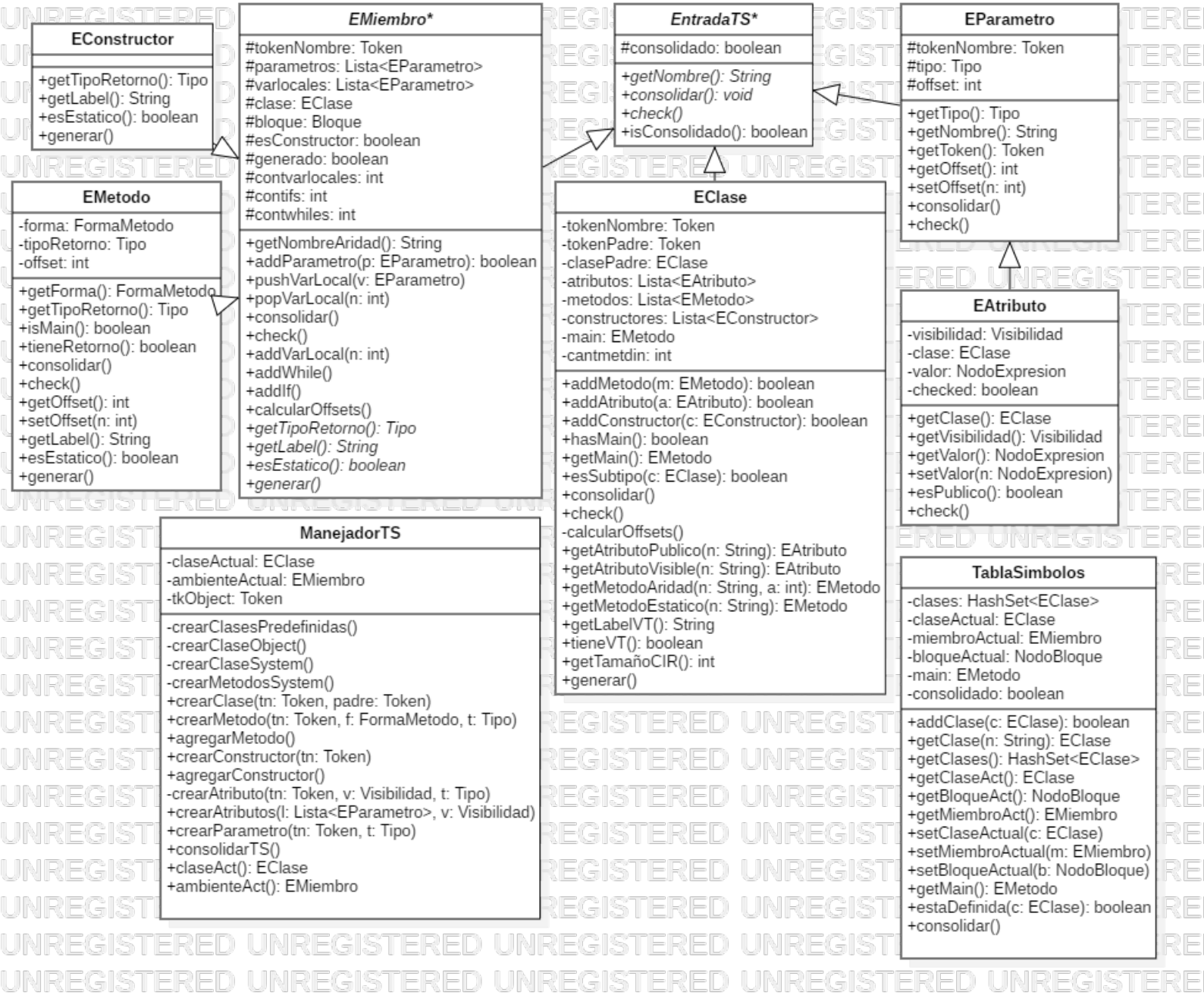
[Expresion {Se setea al NodoExpresion retornado como expresion del NodoArregloEncad}

] Encadenado {Se setea al Encadenado retornado como encadenado del NodoArregloEncad}

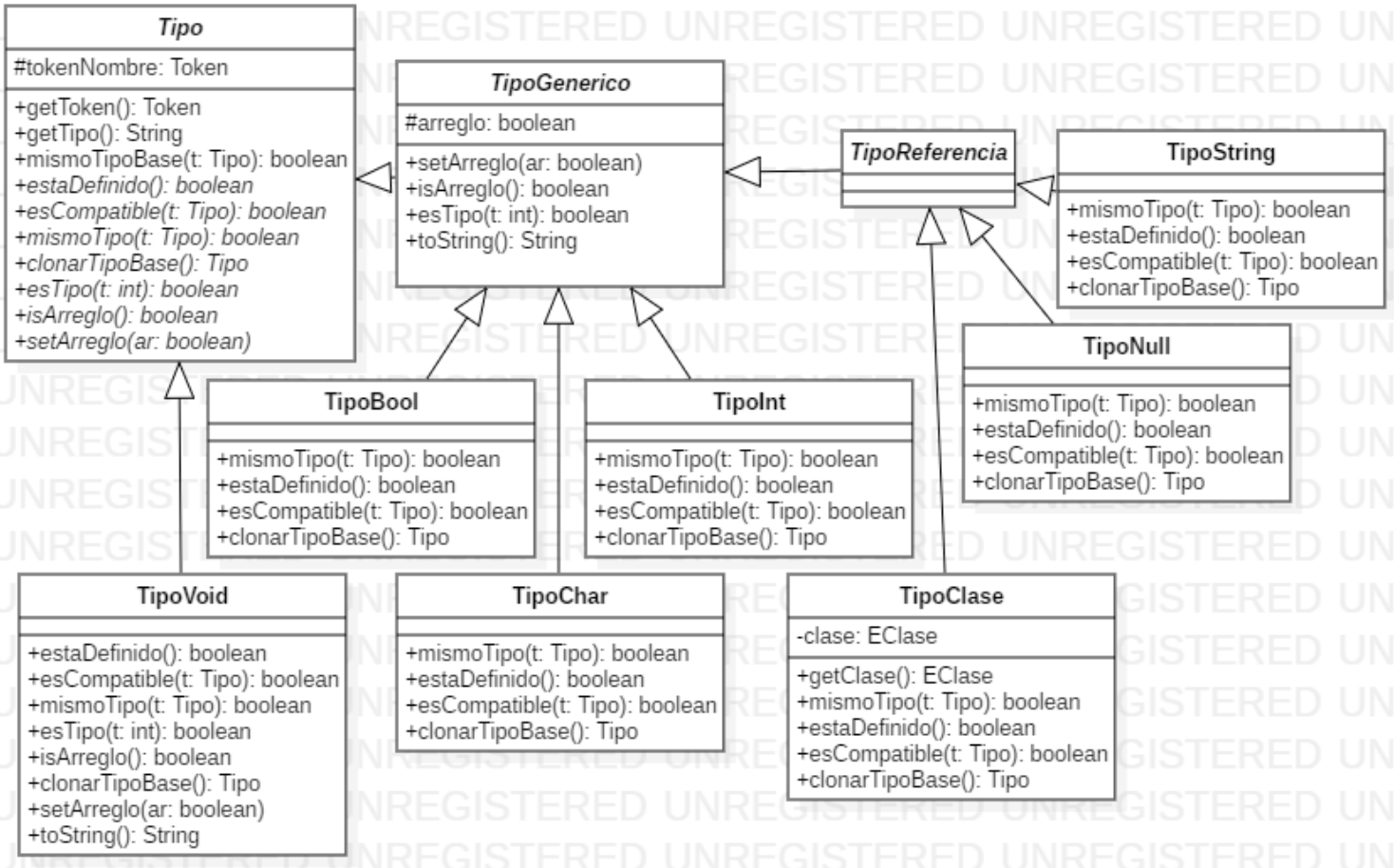
{Se retorna el NodoArregloEncad creado}

Diagramas de clases

Tabla de símbolos:

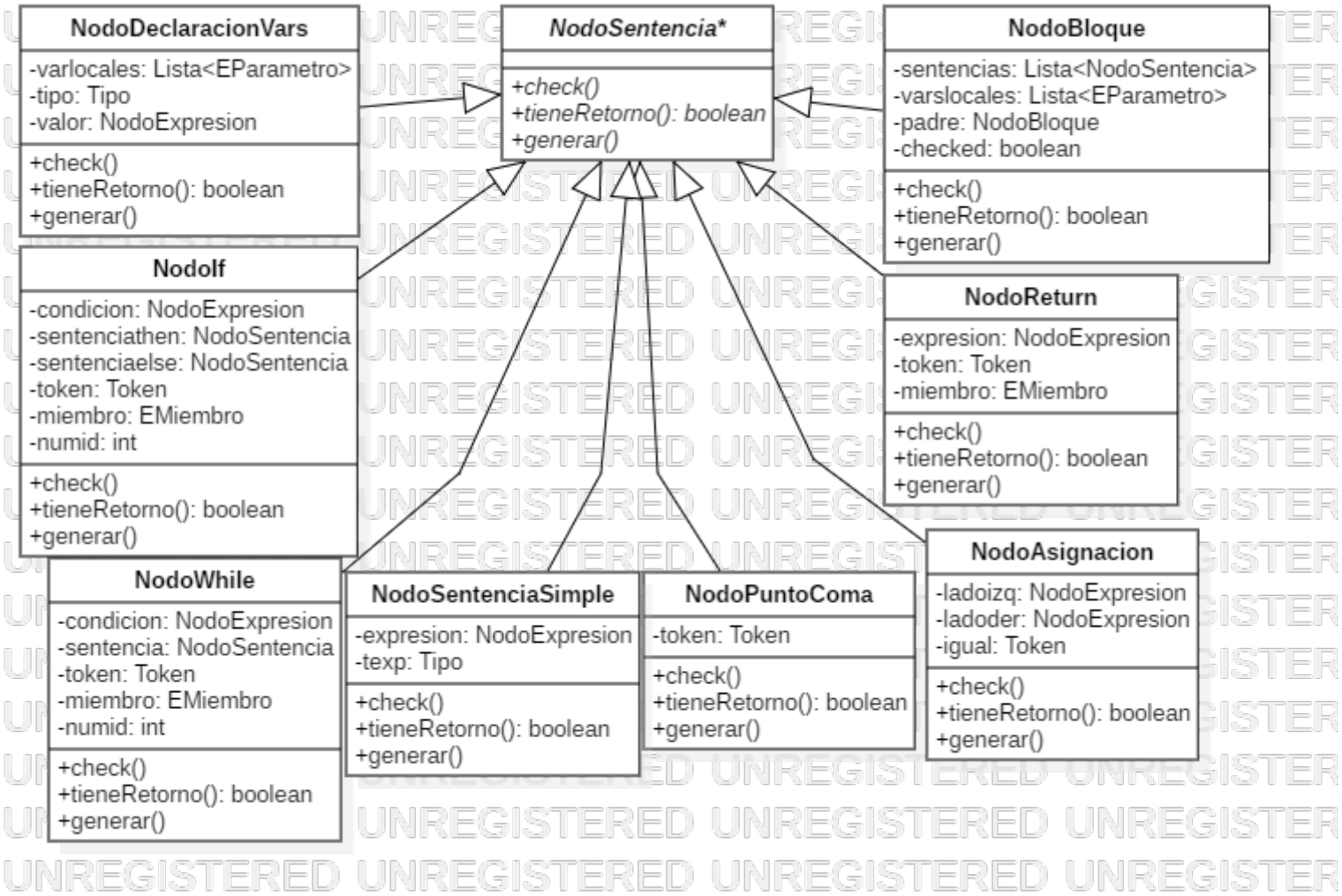


Tipos:

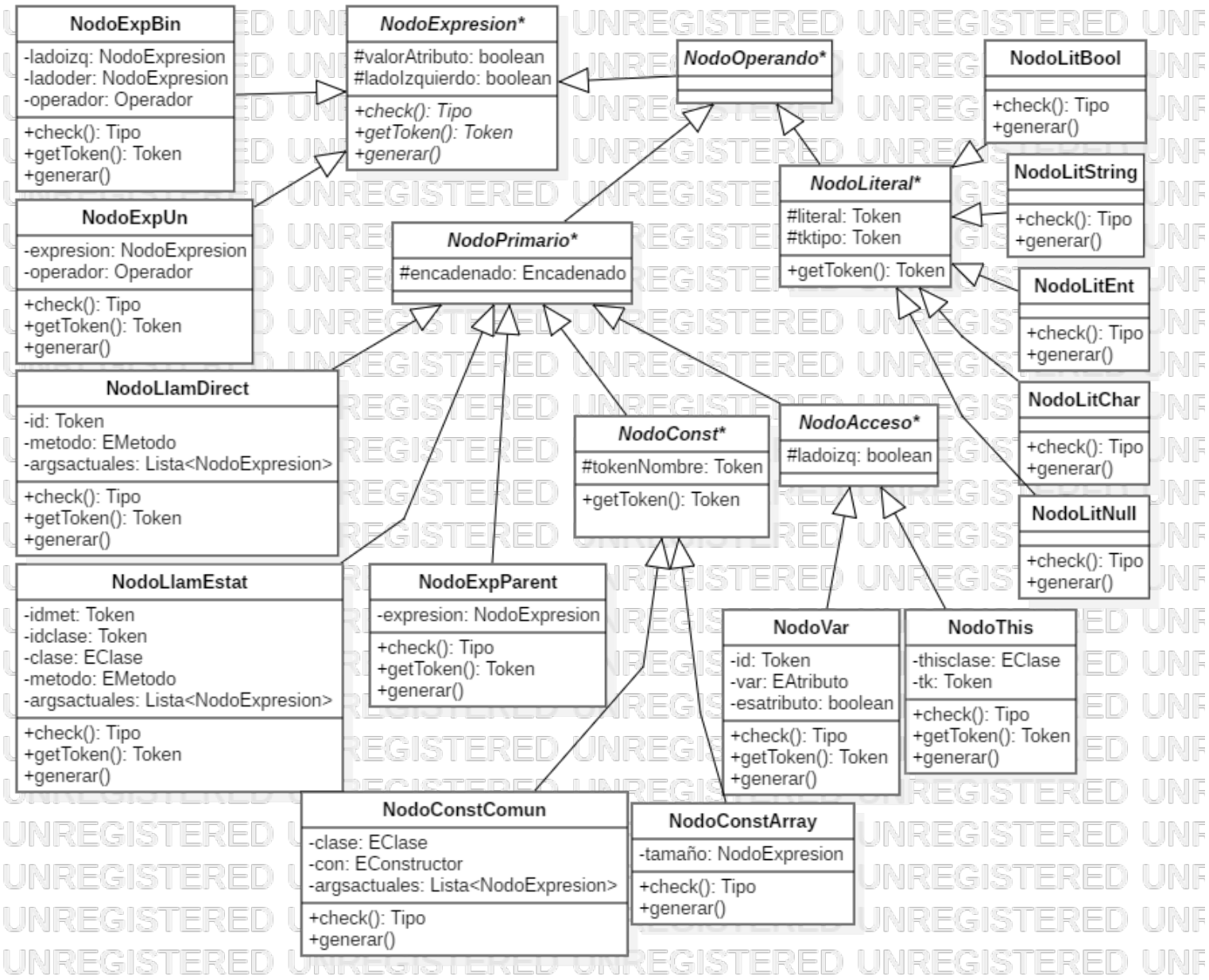

Tipo
#tokenNombre: Token
+getToken(): Token
+getTipo(): String
+mismoTipoBase(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+mismoTipo(t: Tipo): boolean
+clonarTipoBase(): Tipo
+esTipo(t: int): boolean
+isArreglo(): boolean
+setArreglo(ar: boolean)
TipoGenerico
#arreglo: boolean
+setArreglo(ar: boolean)
+isArreglo(): boolean
+esTipo(t: int): boolean
+toString(): String
TipoReferencia
TipoString
+mismoTipo(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+clonarTipoBase(): Tipo
TipoNull
+mismoTipo(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+clonarTipoBase(): Tipo
TipoBool
+mismoTipo(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+clonarTipoBase(): Tipo
TipoInt
+mismoTipo(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+clonarTipoBase(): Tipo
TipoVoid
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+mismoTipo(t: Tipo): boolean
+esTipo(t: int): boolean
+isArreglo(): boolean
+clonarTipoBase(): Tipo
+setArreglo(ar: boolean)
+toString(): String
TipoChar
+mismoTipo(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+clonarTipoBase(): Tipo
TipoClase
-clase: EClase
+getClase(): EClase
+mismoTipo(t: Tipo): boolean
+estaDefinido(): boolean
+esCompatible(t: Tipo): boolean
+clonarTipoBase(): Tipo

Árbol sintáctico abstracto (AST):

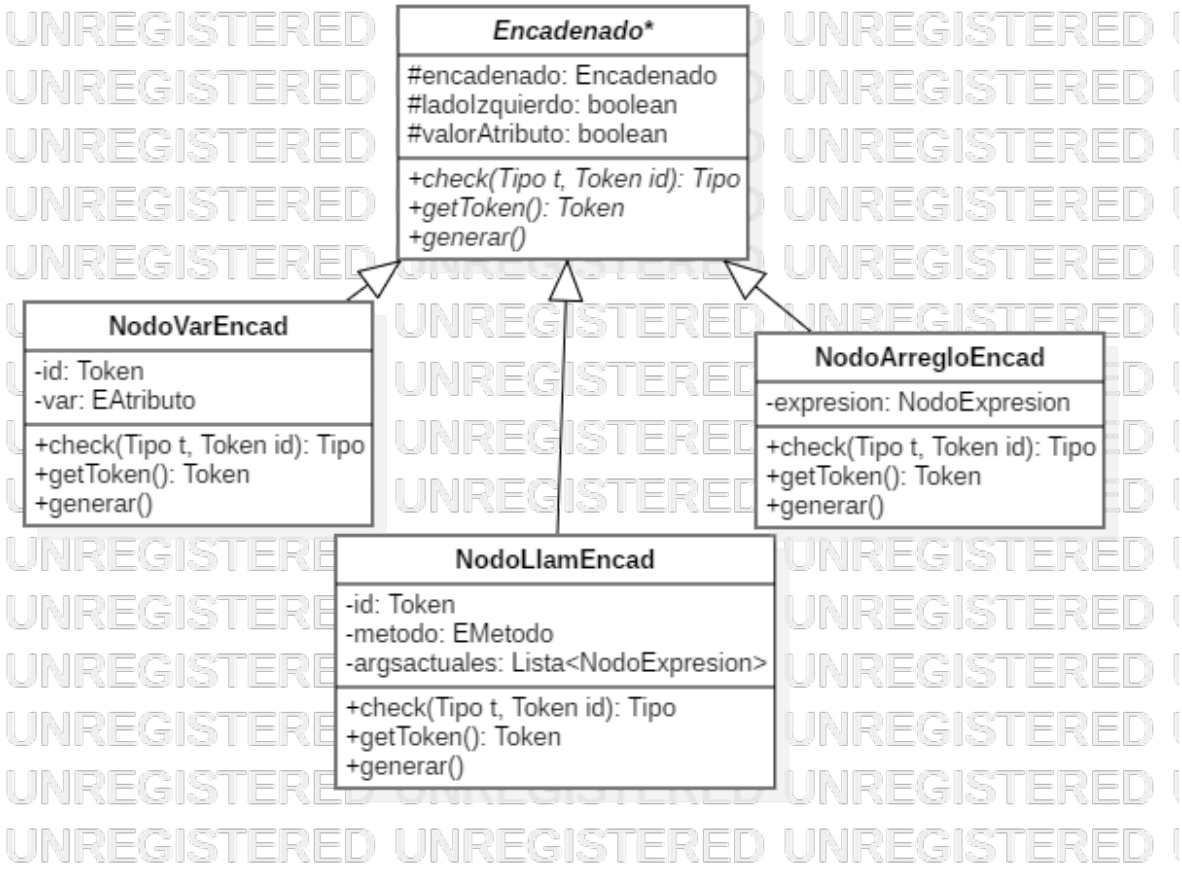
Sentencias:



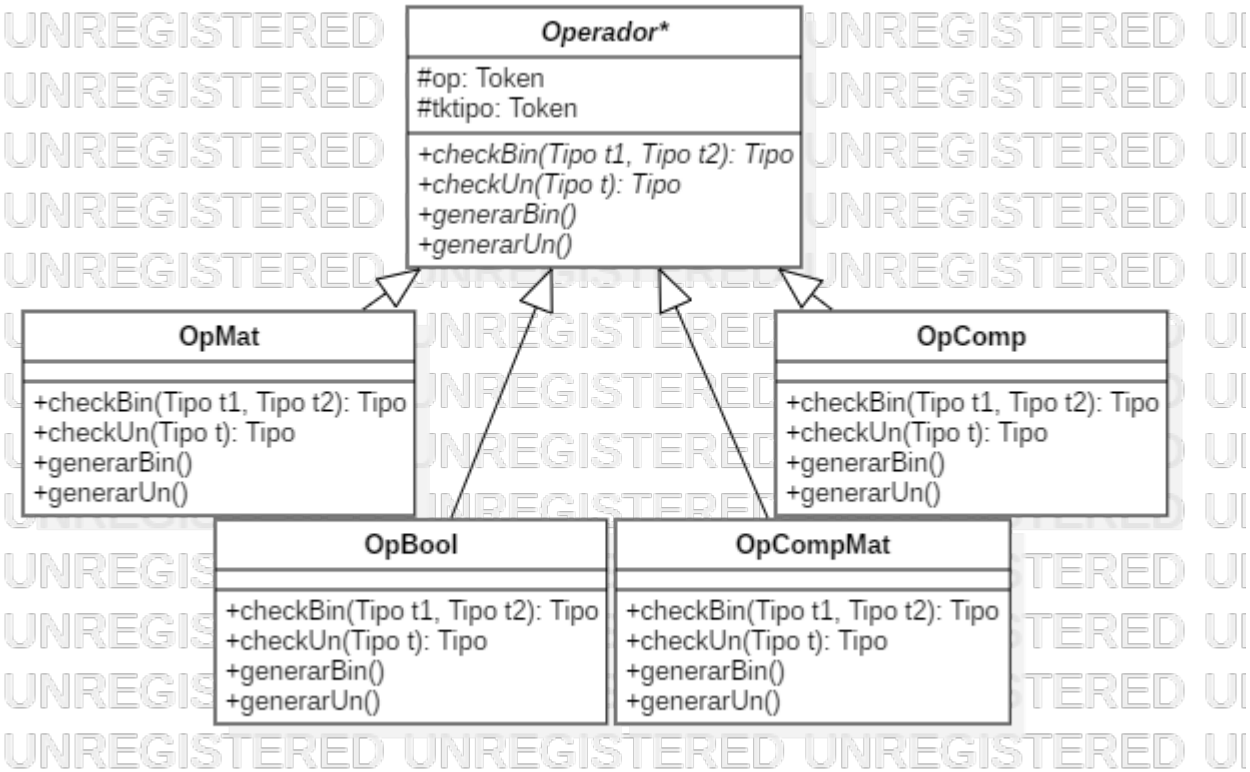
Expresiones:



Encadenados:



Operadores:



Etapa I. Analizador Léxico

Decisiones de diseño:

- Se utiliza el algoritmo “switch case” para la implementación del autómata reconocedor del analizador léxico.

- El autómata que reconoce los distintos tipos de tokens es reducido, es decir, algunos estados reconocen mas de un tipo de token. En particular el estado inicial(0) reconoce todos los tokens de un solo carácter, el resto de los estados reconocen a lo sumo 2 tipos distintos de tokens.

- Las palabras claves son reconocidas dentro de los estados que reconocen los tokens de tipo Identificador de Método-Variable e Identificador de Clase. Se utiliza la un método que provee la clase Utl(Utilidades) que retorna el tipo del token según un lexema.

- Los tipos de los tipos de tokens son enteros, se utilizan constantes para facilitar de lectura/escritura.

Errores detectados.

Caracter inesperado o inválido:
Ocurre cuando el analizador encuentra un carácter que no pertenece al alfabeto o cuando encuentra un carácter que si pertenece a dicho alfabeto pero se encuentra en una ubicación no esperada (por ejemplo un carácter $ fuera de un literal cadena de caracteres).

Literal cadena de caracteres mal formada:
Ocurre cuando un token de tipo literal cadena de caracteres no ha sido cerrado correctamente (por ejemplo se encontró un salto de linea, o el fin del archivo antes de la comilla doble que cierra el literal).

Literal caracter vacío:
Ocurre cuando un token de tipo literal carácter es vacío, es decir cuando se encuentran dos comillas simples seguidas (‘’) o un mal uso del carácter de escape (barra invertida \) (por ejemplo si se encuentra lo siguiente: ‘\’).

Literal caracter mal formado:
Ocurre cuando un token de tipo literal carácter no ha sido cerrado correctamente o cuando contiene mas de un único carácter dentro de las comillas (por ejemplo: ‘aa’ , ‘b , ‘\qq’).

Comentario multilinea sin cierre:
Ocurre cuando un comentario multilinea no es cerrado correctamente con los caracteres ‘*/’.

Operador AND (&&) mal formado:
Ocurre cuando un token de tipo operador AND se encuentra mal formado, es decir luego del carácter & se encuentra otro carácter que no es & o es el fin del archivo.

Operador OR (||) mal formado:
Ocurre cuando un token de tipo operado OR se encuentra mal formado, es decir, luego del carácter | se encuentra otro carácter que no es | o es el fin del archivo.

Etapa II. Analizador Sintáctico I

Decisiones de diseño:

- Se optó por utilizar solo un tipo de excepción para los errores sintácticos y utilizar el mensaje para especificar el tipo de error.

- El tipo de los tokens es un entero, y para compararlos se utilizan métodos provistos por la clase Token.

Errores detectados:

Los errores detectados pueden ser agrupados en dos grupos:

- Errores simples de un token inesperado, que ocurren al intentar hacer un “match” del token actual con el token esperado. Estos errores solo brindan esa información.

- Errores específicos, que son capturados antes de realizar un “match” y brindan mas información sobre que tipo de error, los tokens que eran esperados y el token actual.

A continuación se listaran y explicaran brevemente los errores específicos que el compilador es capaz de detectar.

Error de token no incluido en grupo (Token inesperado):
Este es el error especifico mas simple. Es similar al error simple salvo que brinda la información de varios tipos de tokens que el analizador espera recibir y el token encontrado.

Declaración de clase mal formada:
Este error se da cuando la declaración de una clase esta mal formada, es decir, no se comienza con la palabra reservada class. Brinda información del token esperado y el token encontrado.

Miembro mal formado:
Este error se da cuando un miembro de una clase esta mal formado, es decir, no se encontró un token valido que forme parte del comienzo de un atributo, un método, o un constructor. Brinda información de los tokens esperados y el token encontrado.

Método mal formado. Argumentos inexistentes:
Este error ocurre cuando en la declaración de un método falta la lista de argumentos. Brinda información de los tokens esperados y el token encontrado.

Método mal formado. Tipo de método inexistente:
Este error ocurre cuando en la declaración de un método no se encuentra el tipo. Brinda información de los tokens esperados y el token encontrado.

Lista de argumentos mal formada:
Este error se detecta cuando una lista de argumentos esta mal formada. Brinda información de los tokens esperados y el token encontrado.

Tipo invalido:
Este error ocurre cuando un tipo no es valido. Brinda información de los tokens esperados y el token encontrado.

Declaración mal formada:

Este error ocurre cuando una declaración esta mal formada. Brinda información de los tokens esperados y el token encontrado.

Sentencia mal formada:

Este error se detecta cuando una sentencia se encuentra mal formada. Brinda información de los tokens esperados y el token encontrado.

Expresión mal formada:

Este error se detecta cuando una expresión esta mal formada. Brinda información de los tokens esperados y el token encontrado.

Etapa III. Analizador Sintáctico II

Decisiones de diseño:

- Se optó por mantener a la Tabla de Símbolos como una variable global. Esta se encuentra en la clase Utl (Utilidades) como un atributo estático publico.
- Se utiliza una clase “handler” para manejar a la Tabla de Símbolos que es ManejadorTS. Esta clase se encarga de crear todas las entradas de la TS manteniendo la clase y el ambiente actual.
- Los chequeos semánticos de nombres duplicados de clases, atributos, métodos y parámetros son realizados al momento de insertarlos en la tabla de símbolos.
- La clase ManejadorTS es la encargada de crear las entradas de las clases predefinidas.
- Se optó por representar a los Tipos con un diagrama de clases utilizando herencia.
- La Tabla de Símbolos mantiene una referencia al método main de una de sus entradas de clases. En el caso de que este definido mas de un método main, se la referencia apuntara al último encontrado según el orden en el que quedo organizado el HashSet de clases que la TablaSimbolos mantiene.
- Se optó por utilizar solo un tipo de excepción para los errores semánticos y utilizar el mensaje para especificar el tipo de error.

Chequeos realizados y errores detectados:

A continuación se listaran y explicaran brevemente los chequeos semánticos realizados y las estrategias utilizadas para cada uno.

Nombres duplicados:

El chequeo de nombres duplicados es realizado en la primera pasada al momento de insertar un nuevo elemento en los conjuntos. Para esto se redefinió el método equals de la clase Object que es utilizado por las colecciones para comparar los elementos.
Se considera que dos miembros (métodos o constructores) son iguales si tienen el mismo nombre y aridad.
La clase ManejadorTS es la encargada de lanzar la excepción al encontrarse con un error de nombre duplicado. Se detectan todos los errores de nombres duplicados.

Nombre de constructor distinto de nombre de clase:

Este chequeo también es realizado en la primera pasada y es ManejadorTS la clase responsable de hacerlo. Para ello simplemente compara el nombre del constructor con el nombre de la clase actual.

Declaración de tipo clase definida:

Este chequeo se realiza en la segunda pasada al momento de consolidar la Tabla de Símbolos. La tabla de símbolos le pide a cada una de sus clases que se consolide. El primer chequeo que una clase realiza para su consolidación es si su clase padre está definida. Para esto se utiliza el método getClase(String nombre) de la clase TablaSimbolos. Si la clase padre está definida, se guarda su referencia, si la clase padre no está definida se reporta el error. Para las demás declaraciones de tipo clase, tanto en atributos, retornos de métodos o
argumentos, el chequeo es delegado a la clase Tipo.
Un TipoGenerico siempre está definido, un TipoClase está definido si la clase esta definida. La clase TipoClase utiliza el método estaDefinida(String nombre) de TablaSimbolos para saber si su clase está definida, y en caso de que no lo este, se reporta el error.

Herencia circular:

La herencia circular es chequeada en la consolidación de una clase luego de chequear que su clase padre este definida. Para esto se aprovecha que cada clase tiene una referencia a su clase padre, y se utiliza recursión para recorrer el árbol directo de ancestros hasta llegar a la clase Object o a la clase desde la que se partió.
Si existe herencia circular se reporta el error.

Redefinición de métodos:

El chequeo de la redefinición de métodos se realiza al momento en que se agregan los métodos de una superclase a una subclase cuando se está consolidando una clase. Para el agregado de métodos heredados se utiliza un método privado (addMetodoHeredado) similar al método de agregar un método normal (addMetodo) pero con la diferencia de que si se encuentra un método con nombre y aridad igual dentro de
los métodos de la subclase no se lanza excepción, sino que se pregunta si el método a agregar es exactamente igual, utilizando el método hardEquals de la clase EMetodo, y sino lo es, se lanza excepción normalmente (de nombres duplicados), en el otro caso no se realiza ninguna acción ya que el método está redefinido.

No redefinición de atributos:

El chequeo de la no redefinición de atributos se realiza al momento en que se agregan los atributos de una superclase a una subclase cuando se está consolidando una clase. El chequeo es simplemente antes de agregar el atributo heredado, chequear que no haya nombres duplicados con los atributos de la subclase. En caso de que haya nombres duplicados se reporta el error.

Generación de constructor por defecto:

Es chequeo se realiza al momento de consolidar una clase, antes de agregar los miembros heredados de la superclase. El chequeo simplemente consiste en agregar un nuevo constructor sin parámetros si es que el conjunto de constructores de la clase está vacío.

Existencia de método main:

Este chequeo lo realiza la clase TablaSimbolos en la segunda pasada a medida que va recorriendo sus clases para consolidarlas. El chequeo consiste en preguntarle a cada clase si es que tiene un método main, y en el caso de que si lo tenga se lo obtiene y se guarda la referencia. Al final, luego de recorrer todas las clases, se chequea que al menos exista un método main, y en el caso que ninguna clase tenga uno se reporta el error.

Etapa IV. Analizador Semántico

Decisiones de diseño:

- En la inicialización de un atributo de clase, no es posible utilizar otros atributos o métodos dinámicos de la misma clase, ni la palabra reservada “this” para hacer referencia a la propia clase, es decir, los atributos solo pueden ser inicializados con expresiones, constantes, constructores, métodos estáticos de la misma clase o métodos accedidos a través de un objeto creado.

- La Tabla de Símbolos mantiene una referencia a la clase actual, miembro actual y al bloque actual, que son utilizadas durante los chequeos se sentencias.

- Se optó por utilizar solo un tipo de excepción para los errores semánticos y utilizar el mensaje para especificar el tipo de error.

Chequeos realizados y errores detectados:

A continuación explicaran brevemente los chequeos semánticos para cada tipo de nodo del AST.

Chequeos de sentencias:

NodoBloque

Si el bloque no fue chequeado:
Asignar como bloque actual
Recorrer la lista de sentencias invocando el chequeo de cada una
Desapilar del miembro actual las variables locales del bloque
Asignar como bloque actual al bloque padre
Setear que el bloque fue chequeado

NodoDeclaracionVars

Primero chequear que el tipo este definido, si no lo esta reportar error
Si tiene expresión valor, chequear que los tipos sean compatibles sino reportar error
Si no es una sentencia única (dentro de un if o un while)
Recorrer la lista de variables y agregarlas al miembro actual, si hay variables con mismo nombre reportar error
Si no hay error, agregar las variables al bloque actual

NodoIf

Chequear que la condición sea de tipo booleano, sino reportar error
Chequear la sentencia del then
Si tiene sentencia else, chequear la sentencia else

NodoWhile

Chequear que la condición sea de tipo booleano, sino reportar error
Chequear la sentencia.

NodoReturn

Si el miembro actual no es un constructores
Obtener el tipo de retorno del miembros
Si la expresión no es nula
Chequearla y comparar con el tipo de retorno, si no son compatibles reportar error
Si la expresión es nula y el tipo de retorno no es void reportar error

NodoSentenciaSimple

Chequear la expresión

NodoPuntoComa

No hay nada que chequear

NodoAsignacion

Chequear el lado izquierdo y obtener su tipo
Chequear el lado derecho y obtener su tipo
Si los tipos no son compatibles reportar error
Chequeos de expresiones:

NodoExpBin

Chequear el lado izquierdo y obtener su tipo
Chequear el lado derecho y obtener su tipo
Utilizar el chequeo binario del operador con los tipos obtenidos
El tipo resultante es que retorna el operador

NodoExpUn

Chequear la expresión y obtener su tipo
Utilizar el chequeo unario del operador con el tipo obtenido
El tipo resultante es que retorna el operador

NodoLlamDirect

Chequear que el método sea visible desde la clase actual, si no lo es reportar error
Obtener la forma del método
Si la llamada no esta en la nacionalización de un atributo
Obtener la forma del miembro actual
Si el método es estático ó el miembro actual es dinámico
Chequear que los argumentos sean correctos (cantidad, orden y tipo), si no lo son reportar error
Obtener el tipo de retorno del método.
Si la llamada tiene encadenado
Chequear el encadenado pasandole el tipo de retorno del método y retornar el tipo devuelto
Si no
Retornar el tipo de retorno del método
Sino, reportar error de intento de acceso a método dinámico desde un método estático
Si la llamada esta en una nacionalización de atributo
Si el método es estático
Chequear argumentos, reportando error, obtener tipo retorno del método y retornar el tipo dependiendo si tiene encadenado o no
Si el método es dinámico, reportar error

NodoLlamEstat

Primero chequeo que la clase este definida, sino reportar error
Chequeo que el método exista en la clase, sino reportar error
Luego chequeo que los parámetros sean correctos (cantidad, orden y tipos), si no lo son reportar error
Obtener el tipo de retorno del método
Por ultimo si hay encadenado, chequear el encadenado pasandole como parámetro el tipo de retorno y retornar el tipo que devuelve
Sino hay encadenado retornar el tipo de retorno del método

NodoExpParent

Chequear la expresión y obtener su tipo
Si hay encadenado, chequear el encadenado pasandole como parámetro el tipo de la expresión y retornar el tipo que devuelve.

Sino, retornar el tipo de la expresión

NodoConstArray

Chequear el tamaño y obtener su tipo
Chequear que el tipo de tamaño sea entero, si no lo es reportar error
Si el tipo del arreglo es de tipo clase, chequear que la clase este definida, sino reportar error
Luego, si hay encadenado, chequear el encadenado pasandole como parámetro el tipo del arreglo y retornar el tipo que devuelve.
Si no hay encadenado, retornar el tipo del arreglo

NodoConstComun

Primero chequear que la clase este definida, sino reportar error.
Luego chequear que los parámetros sean correctos (cantidad, orden y tipo) con alguno de los constructores de la clase chequeando cada argumentos, si no son correctos reportar error
Obtener el tipo de retorno del constructor.
Por ultimo, si hay encadenado, chequear el encadenado pasandole como parámetro el tipo de retorno y retornar el tipo que devuelve.
Si no hay encadenado, retornar el tipo de retorno del constructor.

NodoVar

Si esta en una nacionalización de atributos, reportar error
Sino, primero chequear si la variable es una variable local o parámetro
Si no esta
Si el miembro actual es estático, reportar error de acceso a variable a instancia.
Si no, chequear si es un atributo visible de la clase actual y si no lo es por ultimo reportar error
Luego obtener el tipo de la variable
Por ultimo, si hay encadenado, chequear el encadenado pasandole como parámetro el tipo de la variable y retornar el tipo que devuelve.
Si no hay encadenado, retornar el tipo de la variable.

NodoThis

Si esta en una inicialización de atributo, reportar error
Si no esta
Si hay encadenado, chequear el encadenado pasandole como parámetro el tipo de la clase y retornar el tipo que devuelve.
Si no hay encadenado, retornar el tipo de la clase.

NodoLitInt
NodoLitBool
NodoLitChar
NodoLitString
NodoLitNull

Para todos los literales simplemente se retorna el tipo del literal

Chequeos de expresiones:

NodoVarEncad

Primero chequear que el tipo recibido sea de tipo clase, si no lo es reportar error
Luego chequear que la clase este definida, sino reportar error
Chequear que la clase tenga un atributo visible desde la clase actual, sino reportar error
Obtener el tipo del atributo
Finalmente, si hay encadenado, chequear el encadenado pasandole como parámetro el tipo del atributo y retornar el tipo que devuelve.
Si no hay encadenado, retornar el tipo del atributo.

NodoLlamEncad

Primero chequear que el tipo recibido sea de tipo clase, si no lo es reportar error
Luego chequear que la clase este definida, sino reportar error
Chequeo que la clase un método con el mismo nombre, sino reportar error
Chequear que los parámetros sean correctos (cantidad, orden y tipo) chequeando cada argumento, si no son correctos reportar error
Obtener el tipo de retorno del método
Finalmente, si hay encadenado, chequear el encadenado pasandole como parámetro el tipo de retorno del método y retornar el tipo que devuelve.
Si no hay encadenado y se esta en un lado izquierdo de una asignación, reportar error
Sino, retornar el tipo de retorno del método

NodoArregloEncad

Primero chequear que el tipo recibido sea un arreglo, si no lo es reportar error
Luego chequear la expresión y obtener el tipo
Chequear que el tipo de la expresión sea entero, si no lo es reportar error
Finalmente, si hay encadenado, chequear el encadenado pasandole como parámetro el tipo recibido como parámetro y retornar el tipo que devuelve.
Si no hay encadenado, retornar el mismo tipo que el recibido pero que no sea arreglo.

Chequeos de operadores:

OpMat

Siempre se chequea que los tipos recibidos como parámetros para los chequeos binarios y unarios sean de tipo entero. El tipo que retorna es entero.

OpBool

Siempre se chequea que los tipos recibidos como parámetros para los chequeos binarios y unarios sean de tipo booleano. El tipo que retorna es booleano.

OpCompMat

Siempre se chequea que los tipos recibidos como parámetros para el chequeos binarios sean de tipo entero. No se utiliza el chequeo unario. El tipo que retorna es booleano.

OpComp

Se chequean que los tipos recibidos como parámetros para el chequeo binario sean compatibles hacia ambos lados. No se utiliza el chequeo unario. El tipo que retorna es booleano.

Etapa V. Generación de código

Decisiones de diseño:

- Para las variables locales a los bloques, se reserva espacio antes de generar el código del bloque y se libera al final. Se lleva un contador con la cantidad de variables utilizadas en cada bloque de cada método o constructor.

- La inicialización de los atributos de clase se realiza luego de llamar al constructor de la clase y antes de generar el código del bloque del constructor.

- Se optó por no agregar a los métodos estáticos a las VT de las clases. Esto implica que si una clase no posee métodos dinámicos no tendrá una VT.

- Se agregó un nuevo método a la clase System; readI() el cual esta pensado para leer un dígito del 0 al 9 por teclado. Su funcionamiento es idéntico al método read() excepto que este resta 48 (código ascii del carácter 0) al carácter ingresado y lo retorna.

- Los métodos de la clase System son generados para ser llamados como cualquier método estático de clase.

- Se utiliza un *this* ficticio para los métodos estáticos para simplificar el acceso a los parámetros y retorno en los métodos sin importar si son estáticos o dinámicos.

Calculo de offsets y etiquetas:

El calculo de offsets de los métodos dinámicos, atributos de clase y los parámetros de un miembro (constructor o método) se realizan en antes de hacer el chequeo de sentencias y luego de realizar consolidar las declaraciones en cada clase. Las clases de la tabla de símbolos proveen métodos para obtener la etiqueta de su VT, si es que tiene una (una clase puede no tener una VT si solo tiene métodos estáticos), y para obtener el tamaño de su CIR. El calculo de offsets de las variables locales se realiza en el NodoDeclaracionVars luego del chequeo de sentencias, para el calculo se utiliza la cantidad de variables locales hasta el momento que tiene el miembro (constructor o método) y la lista de variables del nodo. Para las etiquetas, cada entrada de la tabla de símbolos provee un método para obtenerlas.

Las etiquetas están construidas de la siguiente manera:

Etiqueta método:

```
<nombreclase>_<nombremetodo>_<aridad>
```

Etiqueta método main:

```
main
```

Etiqueta constructor:

```
<nombreclase>const<aridad>
```

Etiqueta VT de clase:

```
VT_<nombreclase>
```

Etiquetas de sentencias if:

```
else_<numero><etiquetamiembro>
endif_<numero><etiquetamiembro>
```

Donde `<numero>` es un numero identificador único para cada sentencia if dentro del bloque del miembro y `<etiquetamiembro>` es una etiqueta completa del método o constructor al que pertenece.

Etiquetas de sentencias while:

```
while_<numero><etiquetamiembro>
endwhile_<numero><etiquetamiembro>
```

Donde `<numero>` es un numero identificador único para cada sentencia while dentro del bloque del miembro y `<etiquetamiembro>` es una etiqueta completa del método o constructor al que pertenece.

Generación de código:

La generación de código se realiza en cada uno de los métodos generar() de la tabla de símbolos y el AST. Para los nodos que necesitan información adicional para poder generar código (como referencias a métodos, atributos, clases, etc), esta información es obtenida al momento de chequear la sentencia o al momento de crear el nodo.
A continuación se explicara brevemente qué código y como lo genera cada clase que no es trivial

TablaSimbolos

Genera el código de malloc y el inicial que llama al método main y termina.

EClase

Genera el código de la clase System, y la VT de la clase si es que tiene.

EMetodo

Genera el código para finalizar el RA del método, reserva espacio para variables locales y si el bloque del método no tiene un retorno para todos los caminos lo generación.

EConstructor

Genera el código para finalizar el RA del constructor, reserva espacio para variables locales.
Si la clase tiene atributos que deben ser inicializados se genera el código para estos antes de generar el código del bloque.

NodoDeclaracionVars

Genera el código para inicializar las variables si es necesario y setea los offsets de cada una.

NodoReturn

Utiliza una referencia al miembro para calcular el offset del lugar del retorno.

NodoSentenciaSimple

Utiliza una referencia al miembro para saber si el tipo de retorno no es void, hacer un “pop”.

NodoExpBin

Utiliza el método de la clase operador para saber de que operación se trata.

NodoExpUn

Utiliza el método de la clase operador para saber de que operación se trata.

NodoLlamDirect
Utiliza una referencia al método para obtener su etiqueta si es estático u obtener su offset si es dinámico.

NodoLlamEstat
Utiliza una referencia al método para obtener su etiqueta.

NodoConstArray
Usa la subrutina malloc para reservar espacio para el arreglo.

NodoConstComun
Utiliza una referencia al constructor para obtener su etiqueta.

NodoVar
Utiliza una referencia a la variable (atributo, parámetro o variable local) para obtener su offset.

NodoVarEncad
Utiliza una referencia a la variable (atributo, parámetro o variable local) para obtener su offset.

NodoLlamEncad
Utiliza una referencia al método para obtener su etiqueta si es estático u obtener su offset si es dinámico.

OpMat
Genera código del operador según si es un operador binario o unario.

OpBool
Genera código del operador según si es un operador binario o unario.

OpCompMat
Genera el código del operador.

OpComp
Genera el código del operador.

Versión de Java

Se utilizó el IDE Eclipse 4.6.3, con Java 1.8.0_161