Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

Compiladores e Intérpretes

Compilador de MINIJAVA

Manual técnico

Ricardo Ferro Moreno

Universidad Nacional del Sur
Bahía Blanca - 1 de diciembre de 2016

Índice

1. Introducción

En el siguiente documento se darán detalles técnicos del diseño y desarrollo del compilador de MINIJAVA, así como también detalles de las pruebas realizadas a lo largo del proyecto.

MINIJAVA es una versión reducida del lenguaje JAVA. Las especificaciones del lenguaje se pueden encontrar en los apuntes brindados por la cátedra:

- Enunciado general del proyecto
  http://cs.uns.edu.ar/~lc/cei/downloads/Proyecto/Enunciado%20Gral%20del%20Proyecto.pdf
- Sintaxis de MiniJava
  http://cs.uns.edu.ar/~lc/cei/downloads/Proyecto/Sintaxis%20MiniJava.pdf
- Semántica de MiniJava
  http://cs.uns.edu.ar/~lc/cei/downloads/Proyecto/MiniJava-Semantica.pdf

Por lo general, un compilador puede dividirse internamente en cuatro etapas:

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico
- Generación de Código Intermedio

Se suele pensar a estas cuatro etapas como módulos que actúan por separados. Sin embargo, en la práctica, y con el fin de optimizar al compilador para reducir la cantidad de etapas (o "pasadas"), el compilador tendrá dos módulos: uno que se encargará del análisis léxico y generación de tokens a medida que se lee el código fuente, y otro módulo que se encarga de hacer en análisis sintáctico y al mismo tiempo parte del análisis semántico. Además, las estructuras generadas y guardadas por el analizador sintáctico servirán para hacer la otra parte del análisis semántico y además generar el código intermedio.

Para correr el código intermedio (parte *backend* del compilador) se utilizará una máquina virtual CEIVM provista por la cátedra, la cual brinda independencia de la arquitectura, ya que facilita que el código intermedio pueda ser ejecutado en cualquier plataforma que soporte el uso de la máquina virtual.

2. Compilación y Modo de uso

A continuación se darán instrucciones sobre cómo compilar y utilizar el programa. Mayor información sobre el uso del programa se puede encontrar en el **Manual de Usuario**.

2.1. Compilación

El programa fue desarrollado *Java Development Kit 1.8.0_101*, por lo que se sugiere compilar con una versión de JDK superior a 1.8.0.

Para realizar la compilación desde la línea de comandos o terminal, es necesario situarse en la carpeta **src** del código fuente, para luego ejecutar el siguiente comando:

```
javac modulos/Principal.java
```

Con dicho comando se crearán todos los archivos **class** de JAVA.

2.2. Modo de uso

Una vez compilado el programa, se puede ejecutar el mismo mediante la máquina virtual de JAVA, siguiendo con el uso de la línea de comandos o terminal es necesario escribir el siguiente comando:

```
java modulos.Principal <Entrada> [Salida]
```

Donde *Entrada* (parámetro obligatorio) representa la ruta al archivo de entrada a ser compilado, mientras que *Salida* (parámetro opcional) representa la ruta al archivo de salida donde se generará el código intermedio.

En caso de no estar especificado el archivo de salida, el compilador intentará generar un nuevo archivo bajo el mismo nombre del archivo de entrada, agregándole una extensión **ceiasm**. Por ejemplo, si el archivo de entrada es *Programa.java* y no se especifica archivo de salida, el código intermedio será generado en un nuevo archivo *Programa.java.ceiasm* en la misma carpeta donde está siendo ejecutado el compilador.

Una vez generado el código intermedio, para poder ejecutarlo es necesario utilizar la máquina virtual CEIVM brindada por la cátedra de la siguiente manera:

```
java -jar CEIVM.jar <CódigoIntermedio>
```

Donde *CódigoIntermedio* representa la ruta al archivo de código intermedio, generado en el paso anterior, y *CEIVM.jar* el archivo ejecutable de la CEIVM.

3. Analizador Léxico

3.1. Alfabeto de entrada

Como alfabeto de entrada, el compilador reconoce y permite cualquier caracter `Unicode`. Los caracteres especiales como por ejemplo @ (arroba), # (numeral), $ (pesos) se permiten en el código fuente pero no forman parte de tokens válidos. Es decir, pueden aparecer en comentarios o en literales caracteres o strings, sin embargo en cualquier otra parte del programa se detectarán como caracteres inválidos.

3.2. Tokens reconocidos

El módulo de análisis léxico del compilador tiene la capacidad de reconocer y crear distintos tipos de tokens de acuerdo a lo leído en el archivo de entrada. A continuación se listan los distintos tipos de tokens reconocibles, junto con su expresión regular que lo caracteriza, y además un caso de ejemplo válido para dicho lexema. Las expresiones regulares están escritas en un formato comprensible por *java.util.regex.Pattern*.

http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html#sum

Notar que para los nombres de los tokens se usa como prefijo PR para **Palabra Reservada**, P para **Puntuación**, O para **Operador**, y L para **Literal**.
Además, en la tabla figura "EOF" que hace referencia al caracter End of File, símbolo utilizado en los archivos de texto para representar el fin del archivo. Su aparición en la tabla es con el fin de aclarar que para el EOF se guarda un token, al igual que para el resto de los lexemas.

Token	Expresión Regular	Ejemplo
idClase	`[A-Z]+([a-z]|[A-Z]|[0-9]|_)*`	`NombreClase`
IdMetVar	`[a-z]+([a-z]|[A-Z]|[0-9]|_)*`	`nombreMetodo`
L_Entero	`[0-9]+[0-9]*`	`647`
PR_Class	`class`	`class`
PR_Extends	`extends`	`extends`
PR_Static	`static`	`static`
PR_Dynamic	`dynamic`	`dynamic`
PR_Public	`public`	`public`
PR_Private	`private`	`private`
PR_Void	`void`	`void`
PR_Boolean	`boolean`	`boolean`
PR_Char	`char`	`char`
PR_Int	`int`	`int`
PR_String	`String`	`String`
PR_If	`if`	`if`
PR_Else	`else`	`else`
PR_While	`while`	`while`
PR_Return	`return`	`return`
PR_Instanceof	`instanceof`	`instanceof`
PR_This	`this`	`this`
PR_New	`new`	`new`
PR_Null	`null`	`null`
PR_True	`true`	`true`
PR_False	`false`	`false`
P_Parentesis_A	`\(`	`(`
P_Parentesis_C	`\)`	`)`
P_Llave_A	{	{
P_Llave_C	}	}
P_Corchetes_A	`\[`	`[`
P_Corchetes_C	`\]`	`]`
P_Punto	`\.`	`.`
P_Puntocoma	;	;
P_Coma	,	,
O_Mayor	>	>
O_Menor	<	<
O_Not	!	!
O_Asignacion	=	=
O_Comparacion	==	==
O_Mayorigual	>=	>=
O_Menorigual	<=	<=
O_Distinto	!=	!=
O_Suma	+	+
O_Resta	-	-
O_Mult	`\*`	*
O_Div	/	/
O_Mod	%	%
O_Or	`||`	||
O_And	&&	&&
EOF	EOF	EOF
L_String	`\"+.+"\`	`"Hola mundo"`
L_Caracter	`\'+[^\s|^\\]{1}\' | \'\\+[^\s]{1}\'`	'x'

3.3. Clases implementadas

3.3.1. Estructuras

Helper Provee un conjunto de métodos para simplificar el código fuente de algunas estructuras condicionales en el analizador léxico. Se pueden mencionar los siguientes métodos:

- **esLetra(letra)**: retorna `true` si un caracter dado es una letra (a-z ó A-Z).
- **esMayuscula(letra)**: retorna `true` si un caracter dado es una mayúscula (A-Z).
- **esDigito(letra)**: retorna `true` si un caracter dado es un dígito (0-9).
- **esUnderscore(letra)**: retorna `true` si un caracter dado es un guión bajo (_).
- **esSeparador(letra)**: retorna `true` si un caracter dado es un separador. Esto es, un espacio, un *Enter*, una tabulación o un retorno de carro.
- **esEnter(letra)**: retorna `true` si un caracter dado es un *Enter*.
- **esIdentificador(letra)**: retorna `true` si un caracter dado es una letra, un dígito o un underscore. Cualquiera de las tres opciones forman parte de posibles caracteres válidos dentro de un identificador escrito correctamente.
- **esEOF(letra)**: retorna `true` si un caracter dado representa un *End of File*.
- **esComillas(letra)**: retorna `true` si un caracter dado es comillas dobles.
- **esApostrofo(letra)**: retorna `true` si un caracter dado es una comilla simple.

TablaTokens Internamente posee un *HashMap* que representa una tabla con los posibles lexemas y palabras reservadas del lenguaje. La tabla es utilizada por el analizador léxico para buscar fácilmente si un lexema es efectivamente una palabra reservada, y en dicho caso, cuál es el nombre del token que debería llevar. La interfaz es muy simple y posee solamente los siguientes dos métodos:

- **esPalabraReservada(cadena)**: retorna `true` si una cadena de texto es una palabra reservada (también puede ser usada para los símbolos del lenguaje).
- **obtenerTipo(cadena)**: retorna el tipo de token para una determinada cadena. Si la cadena no fuera un token reconocido, devuelve `null`.

Token Estructura de datos simple para guardar un Token. El token posee un nombre (que representa el tipo), un lexema, y el número de línea en donde se encuentra. La inicialización del Token se hace únicamente desde su constructor, por lo que el mismo posee únicamente *getters* (y no *setters*). Los métodos de la estructura son:

- **getNombre()**: devuelve el nombre (tipo) del Token.
- **getLexema()**: devuelve el lexema propiamente dicho.
- **getLinea()**: devuelve el número de línea donde fue encontrado el token.

3.3.2. Módulos

Para ésta etapa, como parte de un proyecto de desarrollo incremental se diseñaron solamente dos módulos, los cuales se detallarán a continuación.

Principal Es quien se encarga de la lógica en el manejo de archivos. Su responsabilidad es la interacción con usuario, revisar que la invocación del programa haya sido realizada correctamente y que los archivos de entrada y/o salida existan, y en caso de resultar algún error al respecto, informarlo.
La clase Principal también se encargará de crear al módulo de analizador léxico para darle las órdenes de ir obteniendo los lexemas.

AnalizadorLexico Su función es hacer el análisis léxico de acuerdo al *buffer* de lectura que se le haya asignado. Posee un único método el cual es el servicio que provee la clase, la de obtener el siguiente token del archivo de entrada.

- **getToken()**: Busca según su posición actual en el *buffer*, formando y retornando un objeto de tipo Token si así fuera posible. En caso de no ser posible, en analizador lanzará una excepción informando el error encontrado.

3.3.3. Excepciones

Para mejorar el manejo de errores se crearon diversas excepciones que se pueden encontrar en el paquete `excepciones.lexicas`. Dichas excepciones son utlizadas para mostrar cualquier tipo de error que surjan en esta etapa de la compilación de un programa.

3.4. Estructura del Analizador Léxico

El Analizador Léxico internamente tiene un *buffer* de lectura, una tabla de tokens (que utilizara para obtener fácilmente si un lexema es una palabra reservada o símbolo del lenguaje), y además guarda el número de línea sobre la que se está realizando la lectura y el caracter actual. Estas últimas dos se utilizan para formar el lexema y saber en qué número de línea se encuentra al momento de requerir un nuevo token.

Cuando se inicia el Analizador Léxico, el caracter actual será el primero del *buffer* de lectura, mientras que se comienza a hacer el análisis desde la línea 1.

Los casos de los símbolos que pueden formar tokens fueron agrupados de la siguiente forma:

- **EOF** { } **[ ] ( ) . , ; + - *** %: La operación para éstos símbolos es automática. Se genera el token correspondiente al símbolo y se consume un caracter para simular situarse en la próxima posición del *buffer*.

- $>$ $<$ **! =**: En este caso, es necesario revisar si el siguiente caracter corresponde a un igual (=). Si así lo fuera, el token corresponde a un mayor/menor igual, o a una comparación. En caso de que no lo sea, se retorna el token de único símbolo mayor/menor estricto, símbolo de negación o de asignación.

- **| &**: Para el caso de los operadores OR y AND, cuando se lee alguno de los símbolos de *vertical bar* o *ampersand*, lo lógico es que a continuación aparezca el mismo símbolo. Si no aparece el mismo símbolo, se detecta un error. En caso de que sí aparezca, se genera y se retorna el token correspondiente al OR o al AND.

- **" (comillas)**: Las comillas forman parte de un literal `String`. Se genera una cadena de texto con todo lo que aparezca entre las comillas que abren, hasta que aparezcan las comillas que cierran. En caso de que aparezca un *Enter* o *End of File* durante la generación de la cadena, se reportará el error.

- **' (apóstrofo)**: El apóstrofo (o comilla simple) es el caso del literal caracter. Para ello, se verificará que o bien tenga la forma `'x'`, ó que tenga la forma `'\x'`.

- **/ (barra diagonal)**: La barra diagonal es uno de los casos especiales en el análisis léxico. Se puede dar el caso de que la barra corresponda al símbolo de división, así como también puede ser la apertura de un comentario simple o multilínea. Recordar que los comentarios no son tokens. Si el analizador se encuentra en caso del comentario (cualquiera de los dos tipos), lo que se intentará hacer es consumir todo el comentario (si se pudiera) y una vez hecho esto, realizar una llamada recursiva para obtener el token que esté siguiente al comentario.

El servicio principal del Analizador Léxico es el método *getToken*, cuyos pasos a la hora de obtener un nuevo token serían los siguientes:

1. Consumir los separadores (si es que hay), hasta situarse en un caracter que no sea un separador.

2. Si el caracter es una letra, recorrer hasta formar la palabra y usar la tabla de tokens para evaluar si es una palabra reservada, un identificador de clase o de método o variable. Armar el token correspondiente y retornarlo.

3. Sino, si el caracter es un dígito, recorrer hasta formar el número y retornar el token de dicho número.

4. Sino, ver a cual de los casos restante corresponde. Para éste punto, si no se entró por el paso 2 ó 3, significa que el token puede corresponder a un símbolo (ya sea de puntuación, operador, comentario o literal caracter ó string). Las acciones de cada símbolo dependen de cada caso, los cuales fueron recién mencionados.

5. En caso de que el símbolo no corresponda al comienzo de un token conocido, reportar error.

3.5. Decisiones sobre el diseño

- Durante el análisis léxico es posible detectar errores sintácticos, pero para mantener la simplicidad del código se posterga la búsqueda de errores sintácticos a la etapa correspondiente.

- En la especificación de MINIJAVA no se aclara lo que sucede con los comentarios de línea simple que terminan con un *End of File*. Por ejemplo, se comenta la última línea de código pero no se apreta *Enter*. El analizador léxico por su naturaleza permite este tipo de comentarios.
- El *End of File* será considerado como un `char` que casteado a `int` es igual a -1.

3.6. Pruebas utilizadas

Para ésta etapa, se diseñaron casos de pruebas para verificar el correcto funcionamiento del módulo del análisis léxico. Los casos se dividieron en conjuntos de clase A (correctos) y clase B (incorrectos).

Los casos de clase A de esta batería de pruebas buscan evaluar que se estén formando bien los tokens. Se hicieron casos para palabras reservadas, comentarios, identificadores, literales `String`, literales `char`, símbolos de puntuación, operadores y *End of File*.

Con los casos de clase B las fuentes poseían errores intencionales para ver si el analizador era capaz de detectarlos. Entre estos errores, se pueden mencionar identificadores inválidos, operadoers AND y OR mal formados, caracteres que no pertenecen al alfabeto, literales `char` mal formados, literales `String` mal formados y comentarios multilínea sin cerrar.

4. Analizador Sintáctico

4.1. Evolución de la gramática

A continuación se presentarán las diferentes fases por las que pasó la gramática brindada por la cátedra, para poder llegar a la gramática sin extensión BNF, ni recursiva a izquierda, factorizada, para poder hacer posible la implementación del analizador sintáctico descendente predictivo recursivo.

4.1.1. Gramática original

La gramática original de MINIJAVA, brindada por la cátedra.

<*Inicial*> → <*Clase*> $^{+}$
<*Clase*> → `class idClase` <*Herencia*>$^{?}$ { <*Miembros*>* }
<*Herencia*> → `extends idClase`
<*Miembro*> → <*Atributo*> | <*Ctor*> | <*Metodo*>
<*Atributo*> → <*Visibilidad*> <*Tipo*> <*ListaDecVars*> ;
<*Metodo*> → <*FormaMetodo*> <*TipoMetodo*> `idMetVar` <*ArgsFormales*> <*Bloque*>
<*Ctor*> → `idClase` <*ArgsFormales*> <*Bloque*>
<*Visibilidad*> → `public | private`
<*ArgsFormales*> → (<*ListaArgsFormales*>$^{?}$)
<*ListaArgsFormales*> → <*ArgFormal*>
<*ListaArgsFormales*> → <*ArgFormal*> , <*ListaArgsFormales*>
<*ArgFormal*> → <*Tipo*> `idMetVar`
<*FormaMetodo*> → `static | dynamic`
<*TipoMetodo*> → <*Tipo*> | `void`
<*Tipo*> → <*TipoPrimitivo*> | `idClase`
<*TipoPrimitivo*> → `boolean | char | int | String`
<*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*> → <*Primario*> = <*Expresion*>
<*SentenciaLlamada*> → <*Primario*>
<*Expresion*> → <*ExpOr*>
<*ExpOr*> → <*ExpOr*> || <*ExpAnd*> | <*ExpAnd*>
<*ExpAnd*> → <*ExpAnd*> && <*ExpIg*> | <*ExpIg*>

<*ExpIg*> → <*ExpIg*> <*OpIg*> <*ExpComp*> | <*ExpComp*>
<*ExpComp*> → <*ExpAd*> <*OpComp*> <*ExpAd*>
<*ExpComp*> → <*ExpAd*> `instanceof idClase`
<*ExpComp*> → <*ExpAd*>
<*ExpAd*> → <*ExpAd*> <*OpAd*> <*ExpMul*> | <*ExpMul*>
<*ExpMul*> → <*ExpMul*> <*OpMul*> <*ExpUn*> | <*ExpUn*>
<*ExpUn*> → <*OpUn*> <*ExpUn*> | <*ExpCast*>
<*ExpCast*> → [`idClase`] <*Operando*> | <*Operando*>
<*OpIg*> → == | !=
<*OpComp*> → < | > | <= | >=
<*OpAd*> → + | -
<*OpUn*> → + | - | !
<*OpMul*> → * | / | %
<*Operando*> → <*Literal*>
<*Operando*> → <*Primario*>
<*Literal*> → `null` | `true` | `false` | `intLiteral` | `charLiteral` | `stringLiteral`
<*Primario*> → (<*Expresion*>) <*LlamadaoIdEncadenado*>*
<*Primario*> → `this` <*LlamadaoIdEncadenado*>*
<*Primario*> → `idMetVar` <*LlamadaoIdEncadenado*>*
<*Primario*> → (`idClase` . <*Llamada*>) <*LlamadaoIdEncadenado*>*
<*Primario*> → `new idClase` <*ArgsActuales*> <*LlamadaoIdEncadenado*>*
<*Primario*> → <*Llamada*> <*LlamadaoIdEncadenado*>*
<*LlamadaoIdEncadenado*> → . <*Llamada*>
<*LlamadaoIdEncadenado*> → . `idMetVar`
<*Llamada*> → `idMetVar` <*ArgsActuales*>
<*ArgsActuales*> → (<*ListaExps*>$^{?}$)
<*ListaExps*> → <*Expresion*>
<*ListaExps*> → <*Expresion*> , <*ListaExps*>

4.1.2. Gramática sin extensión BNF

El primer paso de modificación en la gramática fue eliminarle la extensión BNF (símbolos de + * ?).

<*Inicial*> → <*Clase*> | <*Clase*> <*Inicial*>
<*Clase*> → `class idClase` <*Herencia*> { <*ListaMiembro*> }
<*Herencia*> → `extends idClase` | λ
<*ListaMiembro*> → <*Miembro*> <*ListaMiembro*> | λ
<*Miembro*> → <*Atributo*> | <*Ctor*> | <*Metodo*>
<*Atributo*> → <*Visibilidad*> <*Tipo*> <*ListaDecVars*> ;
<*Metodo*> → <*FormaMetodo*> <*TipoMetodo*> `idMetVar` <*ArgsFormales*> <*Bloque*>
<*Ctor*> → `idClase` <*ArgsFormales*> <*Bloque*>
<*Visibilidad*> → `public` | `private`
<*ArgsFormales*> → (<*LAForm*>)
<*LAForm*> → <*ListaArgsFormales*> | λ
<*ListaArgsFormales*> → <*ArgFormal*>
<*ListaArgsFormales*> → <*ArgFormal*> , <*ListaArgsFormales*>
<*ArgFormal*> → <*Tipo*> `idMetVar`
<*FormaMetodo*> → `static` | `dynamic`
<*TipoMetodo*> → <*Tipo*> | `void`
<*Tipo*> → <*TipoPrimitivo*> | `idClase`
<*TipoPrimitivo*> → `boolean` | `char` | `int` | `String`
<*ListaDecVars*> → `idMetVar`
<*ListaDecVars*> → `idMetVar` , <*ListaDecVars*>
<*Bloque*> → <*ListaSentencias*>
<*ListaSentencias*> → <*Sentencia*> <*ListaSentencias*> | λ
<*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*> `;` | `return ;`
<*Asignacion*> → <*Primario*> = <*Expresion*>
<*SentenciaLlamada*> → <*Primario*>
<*Expresion*> → <*ExpOr*>
<*ExpOr*> → <*ExpOr*> || <*ExpAnd*> | <*ExpAnd*>
<*ExpAnd*> → <*ExpAnd*> && <*ExpIg*> | <*ExpIg*>
<*ExpIg*> → <*ExpIg*> <*OpIg*> <*ExpComp*> | <*ExpComp*>
<*ExpComp*> → <*ExpAd*> <*OpComp*> <*ExpAd*>
<*ExpComp*> → <*ExpAd*> `instanceof idClase`
<*ExpComp*> → <*ExpAd*>
<*ExpAd*> → <*ExpAd*> <*OpAd*> <*ExpMul*> | <*ExpMul*>

<*ExpMul*> → <*ExpMul*> <*OpMul*> <*ExpUn*> | <*ExpUn*>
<*ExpUn*> → <*OpUn*> <*ExpUn*> | <*ExpCast*>
<*ExpCast*> → [`idClase`] <*Operando*> | <*Operando*>
<*OpIg*> → == | !=
<*OpComp*> → < | > | <= | >=
<*OpAd*> → + | -
<*OpUn*> → + | - | !
<*OpMul*> → * | / | %
<*Operando*> → <*Literal*>
<*Operando*> → <*Primario*>
<*Literal*> → `null` | `true` | `false` | `intLiteral` | `charLiteral` | `stringLiteral`
<*Primario*> → (<*Expresion*>) <*ListaLOI*>
<*Primario*> → `this` <*ListaLOI*>
<*Primario*> → `idMetVar` <*ListaLOI*>
<*Primario*> → (`idClase` . <*Llamada*>) <*ListaLOI*>
<*Primario*> → `new idClase` <*ArgsActuales*> <*ListaLOI*>
<*Primario*> → <*Llamada*> <*ListaLOI*>
<*ListaLOI*> → <*LlamadaoIdEncadenado*> <*ListaLOI*> | λ
<*LlamadaoIdEncadenado*> → . <*Llamada*>
<*LlamadaoIdEncadenado*> → . `idMetVar`
<*Llamada*> → `idMetVar` <*ArgsActuales*>
<*ArgsActuales*> → (<*ListaExpsOpc*>)
<*ListaExpsOpc*> → <*ListaExps*> | λ
<*ListaExps*> → <*Expresion*>
<*ListaExps*> → <*Expresion*> , <*ListaExps*>

4.1.3. Gramática sin recursión a izquierda

El siguiente paso de modificación en la gramática fue eliminarle la recursión a izquierda a las reglas que tuvieran esta propiedad.

<*Inicial*> → <*Clase*> | <*Clase*> <*Inicial*>
<*Clase*> → `class idClase` <*Herencia*> { <*ListaMiembro*> }
<*Herencia*> → `extends idClase` | λ
<*ListaMiembro*> → <*Miembro*> <*ListaMiembro*> | λ
<*Miembro*> → <*Atributo*> | <*Ctor*> | <*Metodo*>
<*Atributo*> → <*Visibilidad*> <*Tipo*> <*ListaDecVars*> ;
<*Metodo*> → <*FormaMetodo*> <*TipoMetodo*> `idMetVar` <*ArgsFormales*> <*Bloque*>
<*Ctor*> → `idClase` <*ArgsFormales*> <*Bloque*>
<*Visibilidad*> → `public` | `private`
<*ArgsFormales*> → (<*LAForm*>)
<*LAForm*> → <*ListaArgsFormales*> | λ
<*ListaArgsFormales*> → <*ArgFormal*>
<*ListaArgsFormales*> → <*ArgFormal*> , <*ListaArgsFormales*>
<*ArgFormal*> → <*Tipo*> `idMetVar`
<*FormaMetodo*> → `static` | `dynamic`
<*TipoMetodo*> → <*Tipo*> | `void`
<*Tipo*> → <*TipoPrimitivo*> | `idClase`
<*TipoPrimitivo*> → `boolean` | `char` | `int` | `String`
<*ListaDecVars*> → `idMetVar`
<*ListaDecVars*> → `idMetVar` , <*ListaDecVars*>
<*Bloque*> → <*ListaSentencias*>
<*ListaSentencias*> → <*Sentencia*> <*ListaSentencias*> | λ
<*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*> ; | `return` ;
<*Asignacion*> → <*Primario*> = <*Expresion*>
<*SentenciaLlamada*> → <*Primario*>
<*Expresion*> → <*ExpOr*>
<*ExpOr*> → <*ExpAnd*> <*ExpOrPrimo*>
<*ExpOrPrimo*> → || <*ExpAnd*> <*ExpOrPrimo*> | λ
<*ExpAnd*> → <*ExpIg*> <*ExpAndPrimo*>
<*ExpAndPrimo*> → && <*ExpIg*> <*ExpAndPrimo*> | λ
<*ExpIg*> → <*ExpComp*> <*ExpIgPrimo*>
<*ExpIgPrimo*> → <*OpIg*> <*ExpComp*> <*ExpIgPrimo*> | λ
<*ExpComp*> → <*ExpAd*> <*OpComp*> <*ExpAd*>

<*ExpComp*> → <*ExpAd*> `instanceof idClase`
<*ExpComp*> → <*ExpAd*>
<*ExpAd*> → <*ExpMul*> <*ExpAdPrimo*>
<*ExpAdPrimo*> → <*OpAd*> <*ExpMul*> <*ExpAdPrimo*> | λ
<*ExpMul*> → <*ExpUn*> <*ExpMulPrimo*>
<*ExpMulPrimo*> → <*OpMul*> <*ExpUn*> <*ExpMulPrimo*> | λ
<*ExpUn*> → <*OpUn*> <*ExpUn*> | <*ExpCast*>
<*ExpCast*> → [`idClase`] <*Operando*> | <*Operando*>
<*OpIg*> → == | !=
<*OpComp*> → < | > | <= | >=
<*OpAd*> → + | -
<*OpUn*> → + | - | !
<*OpMul*> → * | / | %
<*Operando*> → <*Literal*>
<*Operando*> → <*Primario*>
<*Literal*> → `null` | `true` | `false` | `intLiteral` | `charLiteral` | `stringLiteral`
<*Primario*> → (<*Expresion*>) <*ListaLOI*>
<*Primario*> → `this` <*ListaLOI*>
<*Primario*> → `idMetVar` <*ListaLOI*>
<*Primario*> → (`idClase` . <*Llamada*>) <*ListaLOI*>
<*Primario*> → `new idClase` <*ArgsActuales*> <*ListaLOI*>
<*Primario*> → <*Llamada*> <*ListaLOI*>
<*ListaLOI*> → <*LlamadaoIdEncadenado*> <*ListaLOI*> | λ
<*LlamadaoIdEncadenado*> → . <*Llamada*>
<*LlamadaoIdEncadenado*> → . `idMetVar`
<*Llamada*> → `idMetVar` <*ArgsActuales*>
<*ArgsActuales*> → (<*ListaExpsOpc*>)
<*ListaExpsOpc*> → <*ListaExps*> | λ
<*ListaExps*> → <*Expresion*>
<*ListaExps*> → <*Expresion*> , <*ListaExps*>

4.1.4. Gramática factorizada

El último paso aplicado sobre la gramática fue la factorización. A continuación se muestra las reglas resultantes luego de la factorización, que son las que se usaron luego para la implementación del analizador sintáctico.

<*Inicial*> → <*Clase*> <*InicialPrimo*>
<*InicialPrimo*> → <*Inicial*> | λ
<*Clase*> → `class idClase` <*Herencia*> { <*ListaMiembro*> }
<*Herencia*> → `extends idClase` | λ
<*ListaMiembro*> → <*Miembro*> <*ListaMiembro*> | λ
<*Miembro*> → <*Atributo*> | <*Ctor*> | <*Metodo*>
<*Atributo*> → <*Visibilidad*> <*Tipo*> <*ListaDecVars*> ;
<*Metodo*> → <*FormaMetodo*> <*TipoMetodo*> `idMetVar` <*ArgsFormales*> <*Bloque*>
<*Ctor*> → `idClase` <*ArgsFormales*> <*Bloque*>
<*Visibilidad*> → `public | private`
<*ArgsFormales*> → (<*LAForm*>
<*LAForm*> → <*ListaArgsFormales*>) |)
<*ListaArgsFormales*> → <*ArgFormal*> <*RestoLAF*>
<*RestoLAF*> → , <*ListaArgsFormales*> | λ
<*ArgFormal*> → <*Tipo*> `idMetVar`
<*FormaMetodo*> → `static | dynamic`
<*TipoMetodo*> → <*Tipo*> | `void`
<*Tipo*> → <*TipoPrimitivo*> | `idClase`
<*TipoPrimitivo*> → `boolean | char | int | String`
<*ListaDecVars*> → `idMetVar` <*RestoLDV*>
<*RestoLDV*> → , <*RestoLDV*> | λ
<*Bloque*> → <*ListaSentencias*>
<*ListaSentencias*> → <*Sentencia*> <*ListaSentencias*> | λ
<*Sentencia*> → ;
<*Sentencia*> → <*AsignacionOSentenciaLlamada*> ;
<*Sentencia*> → <*Tipo*> <*ListaDecVars*> ;
<*Sentencia*> → `if` (<*Expresion*>) <*Sentencia*> <*ElseOpc*>
<*ElseOpc*> → `else` <*Sentencia*> | λ
<*Sentencia*> → `while` (<*Expresion*>) <*Sentencia*>
<*Sentencia*> → <*Bloque*>
<*Sentencia*> → `return` <*ExpresionPrimo*>
<*ExpresionPrimo*> → <*Expresion*> ; | ;
<*AsignacionOSentenciaLlamada*> → = <*Expresion*> | λ
<*Expresion*> → <*ExpOr*>
<*ExpOr*> → <*ExpAnd*> <*ExpOrPrimo*>
<*ExpOrPrimo*> → || <*ExpAnd*> <*ExpOrPrimo*> | λ
<*ExpAnd*> → <*ExpIg*> <*ExpAndPrimo*>
<*ExpAndPrimo*> → && <*ExpIg*> <*ExpAndPrimo*> | λ
<*ExpIg*> → <*ExpComp*> <*ExpIgPrimo*>
<*ExpIgPrimo*> → <*OpIg*> <*ExpComp*> <*ExpIgPrimo*> | λ

<*ExpComp*> → <*ExpAd*> <*OpComp*> <*ExpAd*>
<*ExpComp*> → <*ExpAd*> `instanceof idClase`
<*ExpComp*> → <*ExpAd*>
<*ExpAd*> → <*ExpMul*> <*ExpAdPrimo*>
<*ExpAdPrimo*> → <*OpAd*> <*ExpMul*> <*ExpAdPrimo*> | λ
<*ExpMul*> → <*ExpUn*> <*ExpMulPrimo*>
<*ExpMulPrimo*> → <*OpMul*> <*ExpUn*> <*ExpMulPrimo*> | λ
<*ExpUn*> → <*OpUn*> <*ExpUn*> | <*ExpCast*>
<*ExpCast*> → [`idClase`] <*Operando*> | <*Operando*>
<*OpIg*> → == | !=
<*OpComp*> → < | > | <= | >=
<*OpAd*> → + | -
<*OpUn*> → + | - | !
<*OpMul*> → * | / | %
<*Operando*> → <*Literal*>
<*Operando*> → <*Primario*>
<*Literal*> → `null` | `true` | `false` | `intLiteral` | `charLiteral` | `stringLiteral`
<*Primario*> → (<*PrimarioUno*>
<*PrimarioUno*> → <*Expresion*>) <*ListaLOI*>
<*PrimarioUno*> → `idClase . idMetVar` <*ArgsActuales*>) <*ListaLOI*>
<*Primario*> → `this` <*ListaLOI*>
<*Primario*> → `idMetVar` <*PrimarioDos*>
<*Primario*> → `new idClase` <*ArgsActuales*>) <*ListaLOI*>
<*PrimarioDos*> → <*ListaLOI*> | <*ArgsActuales*> <*ListaLOI*>
<*ListaLOI*> → <*LlamadaoIdEncadenado*> <*ListaLOI*> | λ
<*LlamadaoIdEncadenado*> → `. idMetVar` <*RestoLOI*>
<*RestoLOI*> → <*ArgsActuales*> | λ
<*ArgsActuales*> → (<*ListaExpsOpc*>)
<*ListaExpsOpc*> → <*ListaExps*> | λ
<*ListaExps*> → <*Expresion*> <*RestoListaExps*>
<*RestoListaExps*> → , <*ListaExps*> | λ

4.2. Ambigüedad y LL(1)

La gramática presentada trae una ambigüedad en las siguientes reglas:

<*Sentencia*> → `if` (<*Expresion*>) <*Sentencia*>
<*Sentencia*> → `if` (<*Expresion*>) <*Sentencia*> `else` <*Sentencia*>

Dado que si nos encontramos con la cadena

if (Expresión1) if (Expresión2) Sentencia1 else Sentencia2

Es posible generar dos árboles de derivación distintos, uno en el cual el `else` está asociado al primer `if`, y otro en el cual estará asociado al segundo `if`. Por dicho motivo, la gramática no es LL(1). Aún incluso con la modificación realizada en la gramática, no se soluciona el problema.

<*Sentencia*> → `if` (<*Expresion*>) <*Sentencia*> <*ElseOpc*>
<*ElseOpc*> → `else` <*Sentencia*> | λ

La ambigüedad en este caso se soluciona "automáticamente" a nivel de implementación, por la naturaleza recursiva del analizador. Si hiciéramos una traza con cada instancia del método `sentencia()` en ejecución, observaremos que el método `elseOpc()` será invocado por la instancia que analizó al `if` más reciente. En otras palabras, el `else` se asocia al "último" `if` abierto. Esta solución es la que aplican muchos compiladores en distintos lenguajes de programación.

4.3. Clases Implementadas

4.3.1. Módulos

Principal Para esta etapa, el módulo Principal fue adaptado a las necesidades. Si bien en la etapa previa creaba un objeto de tipo AnalizadorLexico, para ésta etapa crea un objeto AnalizadorSintactico. La lógica del módulo principal sigue siendo la misma, la de interactuar con el usuario

AnalizadorSintactico Su función es hacer el análisis sintáctico. El analizador trabaja de forma predictiva, recursiva y descendente. Internamente tiene un analizador léxico, el cual le va pidiendo tokens a medida de que se necesiten, y además guarda dos tokens, el "actual" (*lookAhead*) y una referencia al "anterior" (*lookBehind*). Este último token sirve de ayuda para dar más exactitud en cuanto a la línea donde se produzca un error sintáctico, en caso de haberlo.

- **analizar()**: Es el único método público que posee el analizador sintáctico, que sirve para dar comienzo al análisis sintáctico de manera completa. En caso de que exista un error sintáctico, se lanzará una excepción y se informará el error. Naturalmente, como utiliza al analizador léxico, también puede lanzar excepciones de este tipo.

4.3.2. Excepciones

Para mejorar el manejo de errores se crearon diversas excepciones que se pueden encontrar en el paquete `excepciones.sintacticas`. Dichas excepciones son utlizadas para mostrar cualquier tipo de error que surjan en esta etapa de la compilación de un programa.

4.4. Estructura del Analizador Sintáctico

El analizador sintáctico guarda internamente un objeto de tipo `AnalizadorLexico`, y dos tokens, uno que servirá como referencia al token actual (*lookAhead*) para comparar el token próximo, y otro token que será referencia al último token procesado (*lookBehind*). Éste último servirá particularmente para intentar mejorar la calidad de los mensajes de error.

Cuando se inicia el Analizador Sintáctico, se crea el Analizador Léxico a partir del *buffer* de lectura que fue dado como parámetro en el constructor, y al mismo tiempo se inicializa el token `lookAhead` para tener uan referencia al primer lexema del archivo de entrada, previo a realizar todo el análisis sintáctico.

Se posee de un único método público `analizar()`, el cual se encargará de hacer todo el análisis sintáctico de la entrada dada. Si existiera algún error durante el análisis, el método lanzará una excepción ya sea léxica o sintáctica.

4.5. Decisiones sobre el diseño

- El *End of File* no forma parte de la gramática, pero su token es utilizado para saber cuándo terminar de analizar las reglas de producción de la gramática. (Revisar código del método `inicialPrimo()`)
- Para el análisis sintáctico predictivo, los *siguientes* y *últimos* son revisados mediante un `switch-case` de JAVA y no con `if-else`. El objetivo de esto es facilitar la lectura del código fuente del módulo.
- Vale la aclaración que no todas las versiones de JAVA permiten utilizar `String` en la estructura de control `switch-case`. Se recomienda utilizar JavaSE 1.7 como mínimo para compilar el proyecto.

4.6. Pruebas utilizadas

Los casos de prueba para esta etapa se dividieron igual que la batería de la etapa anterior. Un conjunto de pruebas de clase A (correctas), y un conjunto de pruebas de clase B (incorrectas). Lógicamente, para esta etapa se busca probar el correcto funcionamiento del analizador sintáctico, por lo que las pruebas de la etapa anterior en su mayoría no se han podido reutilizar.

En cuanto a las pruebas de clase A, se crearon pruebas para los diferentes constuctores del lenguaje. Entre las pruebas, se pueden destacar conjunto de clases simples

(con los requisitos mínimos para ser sintácticamente correctas). Se probó el funcionamiento de constructores, métodos sin argumentos, métodos con uno o más argumentos, declaración de variables locales, declaración de variables de instancia, sentencias de llamada, de `return`, sentencias `if` y `while` simples y anidados, expresiones, asignaciones literales, llamadas en cadena, expresiones de casting e `instanceof`.

Por el lado de las pruebas de clase B, se trató de realizar diferentes clases para verificar la conducta del analizador ante la presencia de errores sintácticos. Por ejemplo, bloques de clases o métodos que no han sido cerrados correctamente, listas de argumentos mal formados, expresiones erróneas, sentencias `if` o `while` mal formadas tanto en la condición como en la sentencia.

5. Analizador Semántico

5.1. Clases Implementadas

5.1.1. Estructuras

TablaSimbolos Para guardar todas las estructuras se posee una tabla de símbolos, la cual posee un conjunto de clases. La tabla de símbolos sirve principalmente para representar las estructuras del programa de entrada del compilador, sin embargo la propia tabla de símbolos realiza chequeos y tareas sobre su propia estructura. Los métodos públicos que posee la clase son:

- **agregarClase(nombre, clase)**: Agrega un nombre y clase dados a la tabla.
- **estaClase(nombre)**: Retorna `true` si la tabla contiene a una clase con un determinado nombre.
- **chequearDeclaraciones()**: Chequea la correctitud semántica de las declaraciones de cada una de sus clases.
- **chequearSentencias()**: Chequea la correctitud semántica de las sentencias que contienen cada una de sus clases.

Clase Sirve para representar cada clase declarada en el código fuente a compilar. Las clases pueden poseer un conjunto de métodos, un conjunto de atributos y un constructor. Además, cada clase sabe el nombre de su clase padre. Si no tuviera clase padre, recordar que todas las clases en `MiniJava` heredan por defecto de `Object`. La responsabilidad de la clase es almacenar las estructuras subyacentes para poder pedirle a cada objeto que realice su chequeo correspondiente, al momento de hacer el chequeo de declaraciones y sentencias. Por otro lado, la propia clase provee dos servicios para `agregarAtributo` y `agregarMetodo` que realizan automáticamente un chequeo de declaraciones para evitar nombres duplicados.

Unidad Para los constructores y métodos se creó una clase `Unidad` que reúne las características que tienen en común, y luego se especializó cada clase utilizando `Metodo` y `Constructor`. Las unidades mantienen un bloque asociado a la unidad, un conjunto de parámetros y variables locales. Los métodos, además de estas características también guardan el tipo de retorno y la forma del método (dinámico o estático). Básicamente la responsabilidad de la clase al momento de hacer sus chequeos es hacer uso de las estructuras que guarda para revisar que no existan los errores semánticos.

Tipos Para representar los tipos, tanto primitivos como definidos por el usuario, se creó el paquete `estructuras.ts.tipos` con sus respectivas clases internamente. Es recomendable revisar la jerarquía de clases en el diagrama de clases mostrado en este apunte.

Variables Tanto variables de instancia (atributos), como parámetros o variables locales son implementadas en el paquete `estructuras.ts.variables`. Si bien las variables poseen cualidades similares, se decidió separarlas en distintos tipos. La que es un tipo especial de las variables son los atributos (variables de instancia), para los cuales además del tipo y el identificador, es necesario guardar si es público o privado.

Expresiones Para representar diferentes tipos de expresiones (incluyendo a las primarias) se creó el paquete `estructuras.ast.expresiones` el cual refleja del lenguaje lo siguiente:

- Expresiones binarias (incluido el instanceof): `NodoExpresionBinaria`
- Expresiones unarias (incluido casteo): `NodoExpresionUnaria`
- Expresiones parentizadas: `NodoExpParentizada`
- Literales: `NodoLiteral`
- Constructores (llamadas a `new`): `NodoConstructor`
- Referencias al objeto actual (llamadas a `this`): `NodoThis`
- Expresiones utilizando variables: `NodoVariable`
- Llamadas a métodos estáticos: `NodoLlamadaEstatica`
- Llamadas a métodos que se resuelven dinámicamente: `NodoLlamadaDirecta`

Sentencias Se crearon clases para representar los distintos tipos de sentencias. Las mismas se encuentran en el paquete `estructuras.ast.sentencias`

- Asignación: `NodoAsignacion`
- Bloque: `NodoBloque`
- Sentencias de retorno: `NodoReturn`
- Declaración de variables locales: `NodoDeclaracion`
- Sentencia vacía: `NodoSentenciaVacia`
- Sentencia de llamada: `NodoSentenciaLlamada`
- Estructura de repetición `while`: `NodoWhile`
- Estructura condicional `if`: `NodoIf`

Encadenados Para las llamadas encadenadas mediante expresiones puntos se crearon las clases `NodoEncadenado` y sus especializaciones `NodoLlamadaEncadenada` y `NodoIdEncadenado`. Al igual que las clases mencionadas anteriormente, estas clases sirven para representar una estructura de una sentencia que posee llamados encadenados.

5.1.2. Modulos

AnalizadorSintactico El módulo de análisis sintáctico sufre una modificación, ya que durante el análisis sintáctico se crearán las estructuras correspondientes a la tabla de símbolos (TS) y árbol de sintaxis abstracta (AST) que serán útiles luego para hacer los chequeos semánticos.

5.1.3. Excepciones

Para mejorar el manejo de errores se crearon diversas excepciones que se pueden encontrar en el paquete `excepciones.semanticas`. Dichas excepciones son utlizadas para mostrar cualquier tipo de error que surjan en esta etapa de la compilación de un programa.

5.2. Estructura del Analizador Semántico

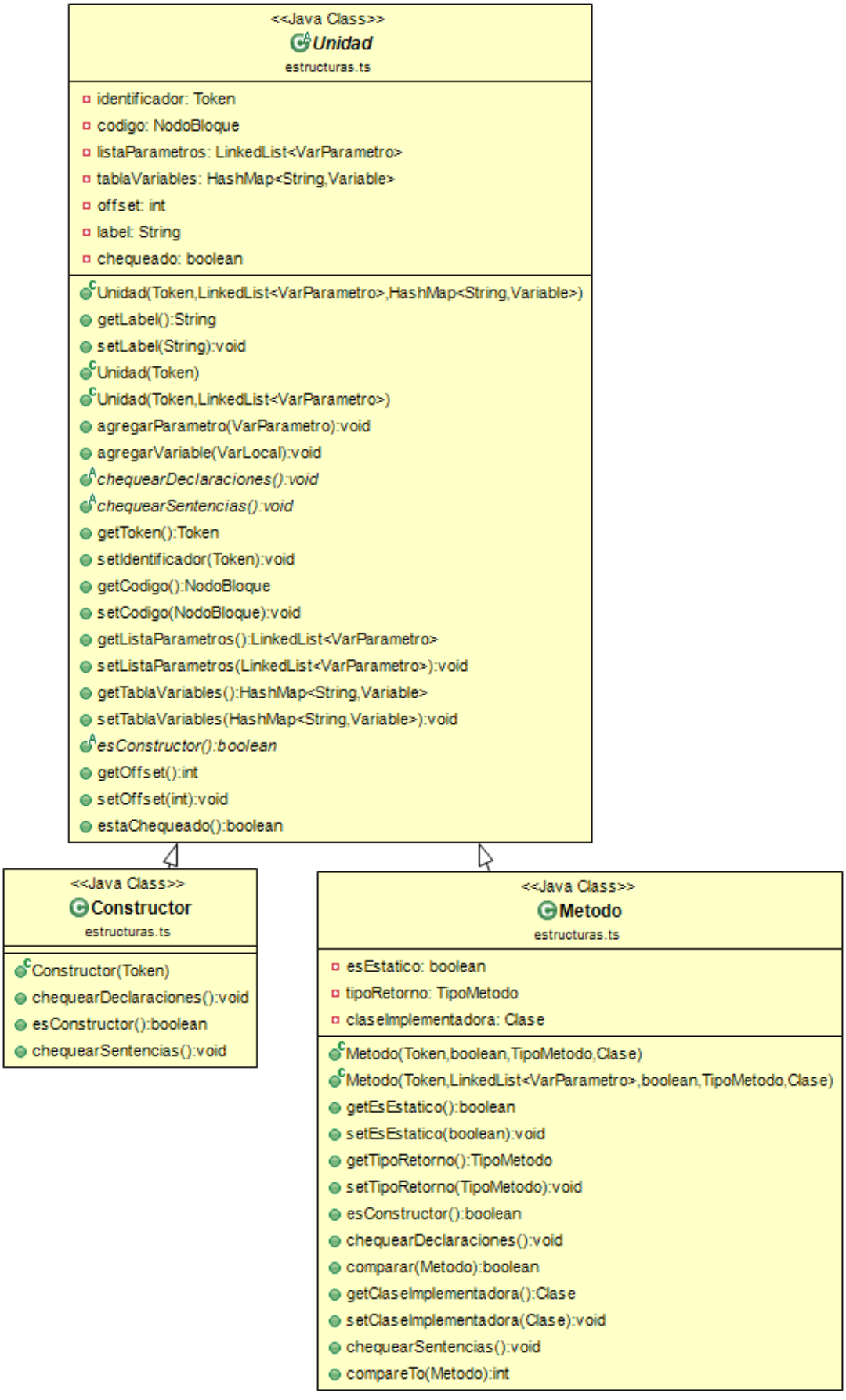
Para que el compilador sea de *dos pasadas* (o dos y media), el analizador semántico es implementado sobre el módulo del analizador sintáctico.

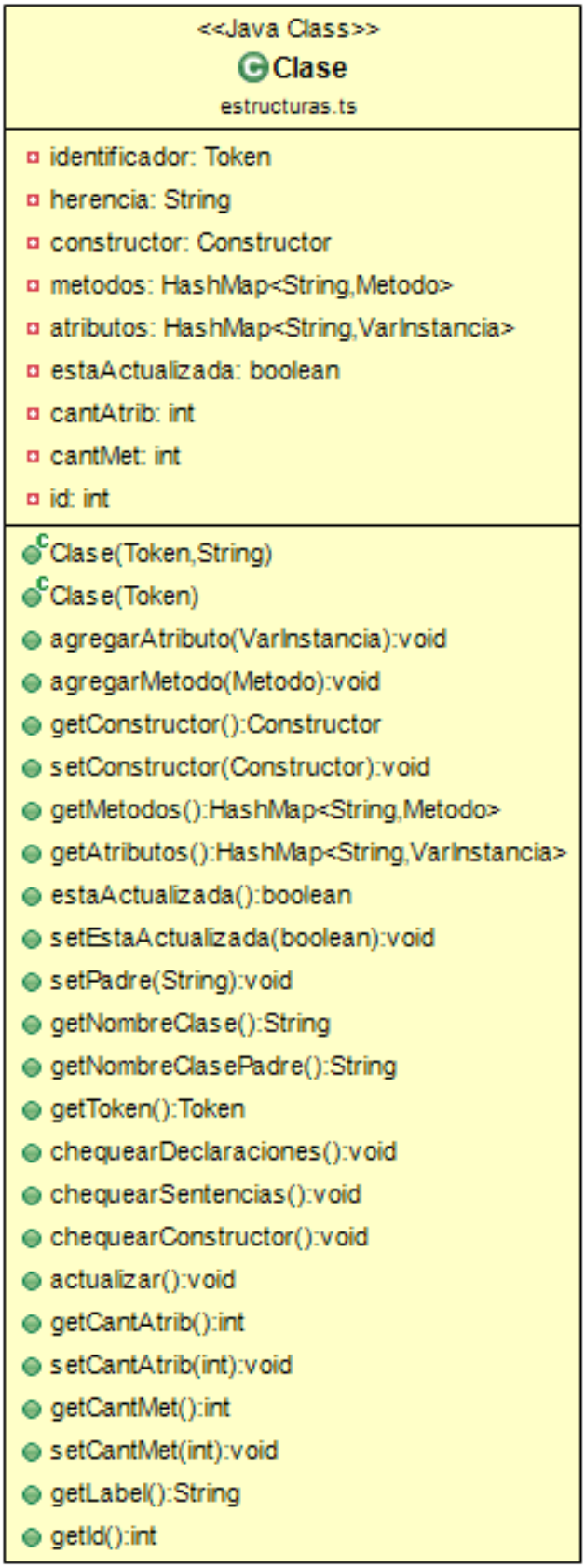
Durante el análisis sintáctico se van creando diferentes nodos que corresponden a la estructura de la tabla de símbolos (TS) y del árbol de sintaxis abstracta (AST). Una vez finalizada la etapa sintáctica, el mismo módulo de análisis sintáctico se encarga de realizar el chequeo de declaraciones y de sentencias sobre la tabla de símbolos.

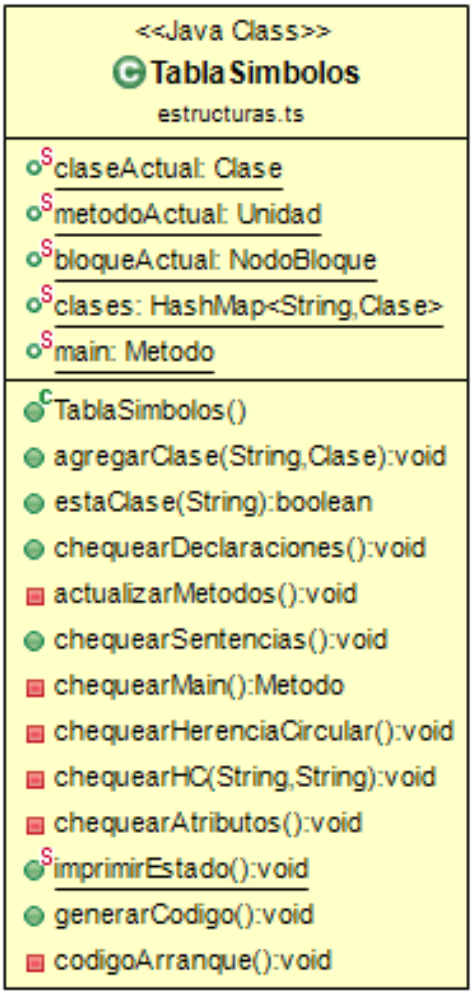
En realidad lo que sucede en cuanto a las responsabilidades, es que el analizador sintáctico (y semántico) hará uso de dos métodos (servicios) provistos por la tabla de símbolos, y la misma tabla de símbolos hará sus propios chequeos internos, obligando a que las clases que posean se verifiquen autónomamente, y así metiéndose en más profundidad. En otras palabras, cada nodo de la tabla de símbolos y AST se encarga de hacer sus chequeos de declaraciones y sentencias como tarea propia, pero quien da inicio a los chequeos es el módulo de análisis sintáctico-semántico una vez finalizada la verificación de correctitud semántica del programa.

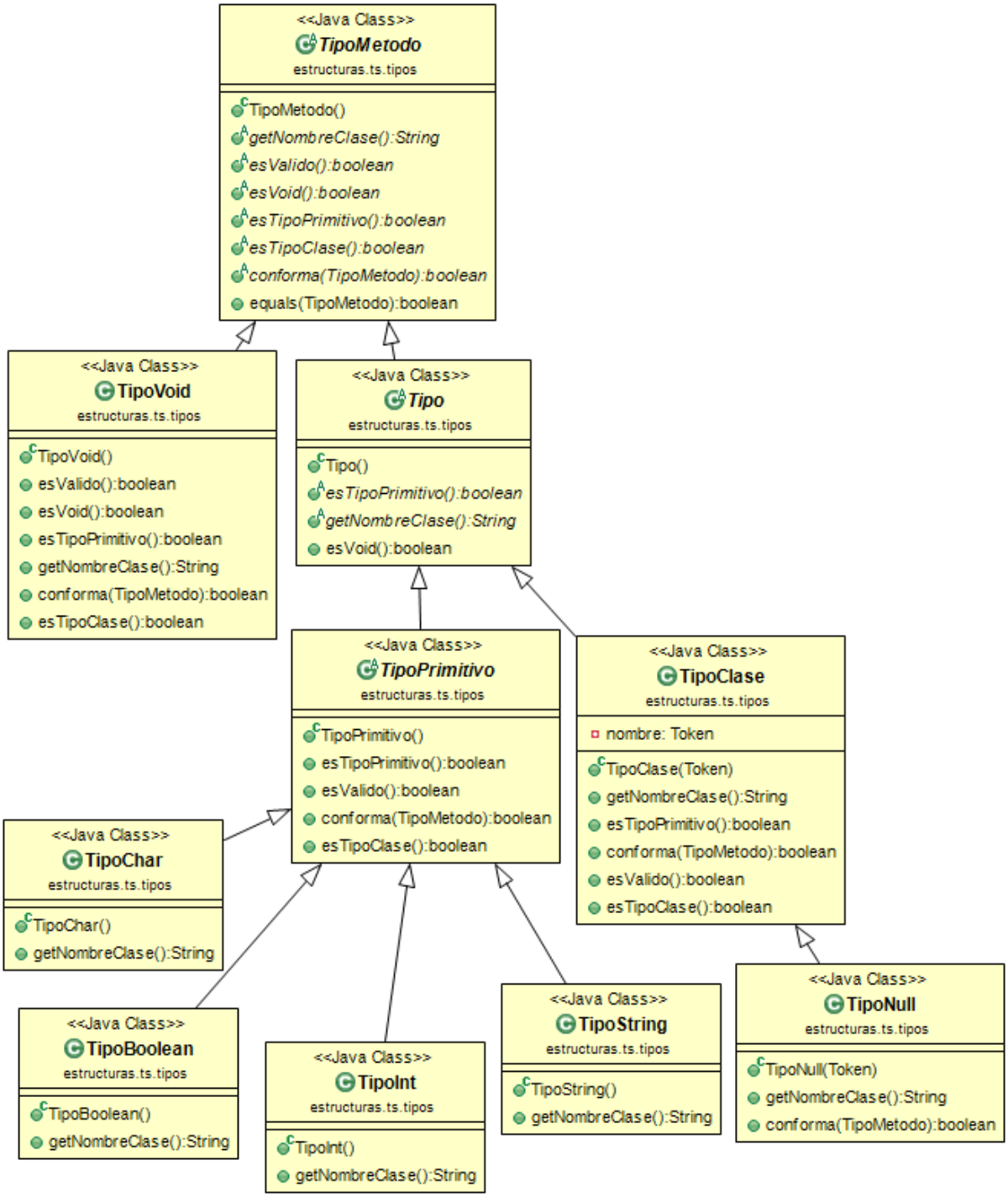
5.3. Jerarquía de TS y AST

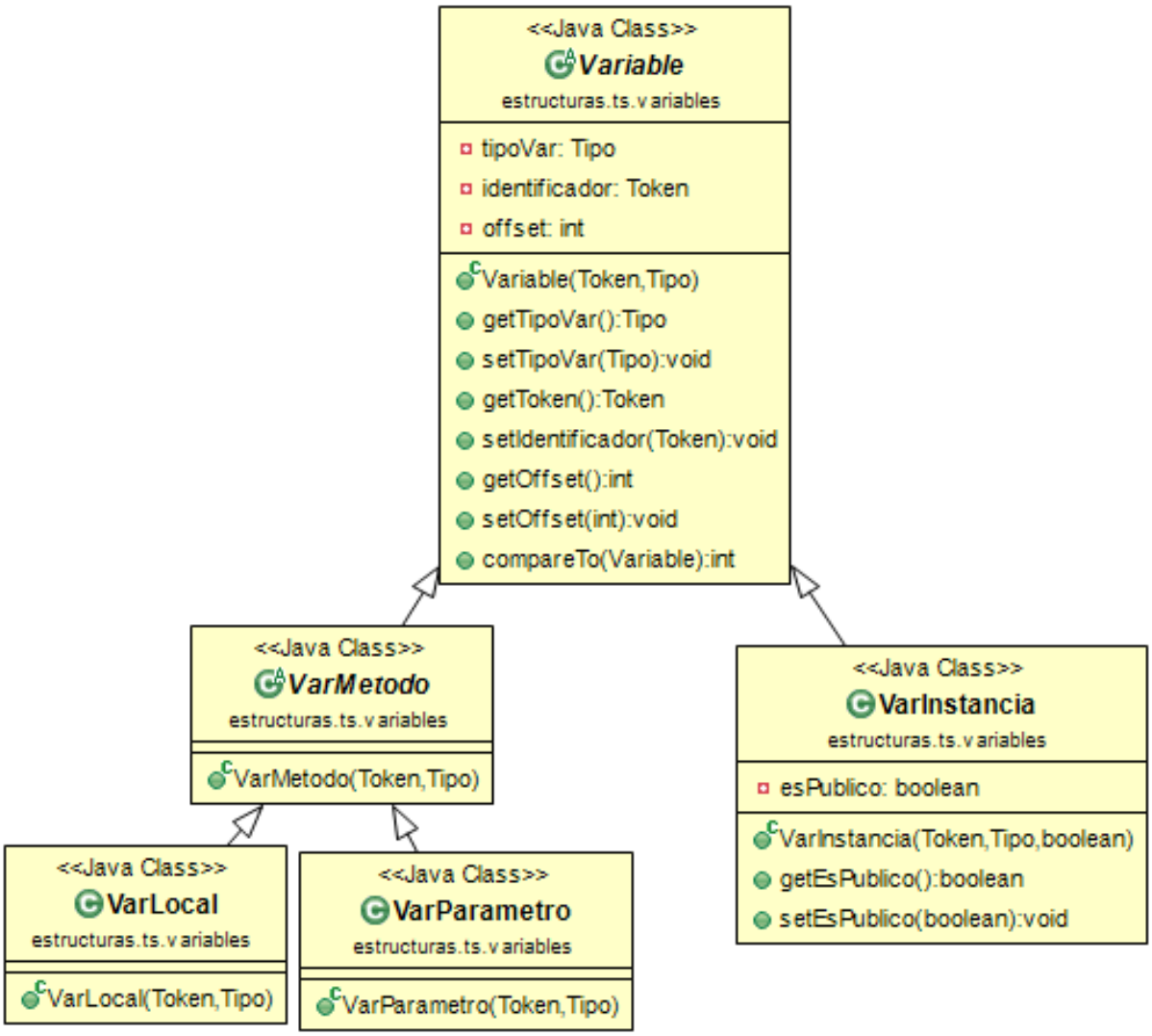
A continuación se ilustra las estructuras utilizadas para la tabla de símbolos y al árbol de sintaxis abstracta. Se presentan en imágenes separadas dado el tamaño de la jerarquía.

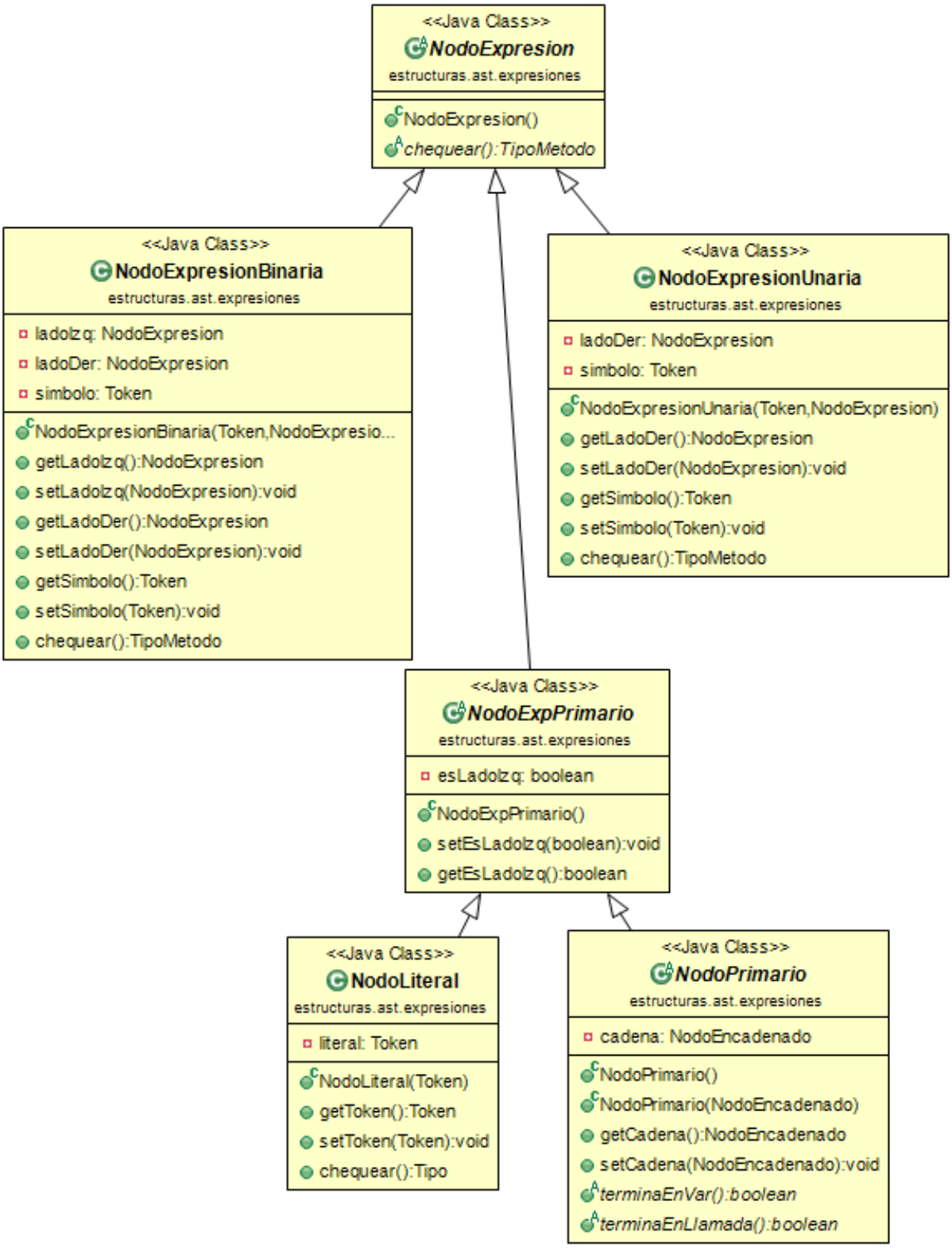

<<Java Class>>
Unidad
estructuras.ts
identificador: Token
codigo: NodoBloque
listaParametros: LinkedList<VarParametro>
tablaVariables: HashMap<String,Variable>
offset: int
label: String
chequeado: boolean
Unidad(Token,LinkedList<VarParametro>,HashMap<String,Variable>)
getLabel():String
setLabel(String):void
Unidad(Token)
Unidad(Token,LinkedList<VarParametro>)
agregarParametro(VarParametro):void
agregarVariable(VarLocal):void
chequearDeclaraciones():void
chequearSentencias():void
getToken():Token
setIdentificador(Token):void
getCodigo():NodoBloque
setCodigo(NodoBloque):void
getListaParametros():LinkedList<VarParametro>
setListaParametros(LinkedList<VarParametro>):void
getTablaVariables():HashMap<String,Variable>
setTablaVariables(HashMap<String,Variable>):void
esConstructor():boolean
getOffset():int
setOffset(int):void
estaChequeado():boolean
<<Java Class>>
Constructor
estructuras.ts
Constructor(Token)
chequearDeclaraciones():void
esConstructor():boolean
chequearSentencias():void
<<Java Class>>
Metodo
estructuras.ts
esEstatico: boolean
tipoRetorno: TipoMetodo
claseImplementadora: Clase
Metodo(Token,boolean,TipoMetodo,Clase)
Metodo(Token,LinkedList<VarParametro>,boolean,TipoMetodo,Clase)
getEsEstatico():boolean
setEsEstatico(boolean):void
getTipoRetorno():TipoMetodo
setTipoRetorno(TipoMetodo):void
esConstructor():boolean
chequearDeclaraciones():void
comparar(Metodo):boolean
getClaseImplementadora():Clase
setClaseImplementadora(Clase):void
chequearSentencias():void
compareTo(Metodo):int

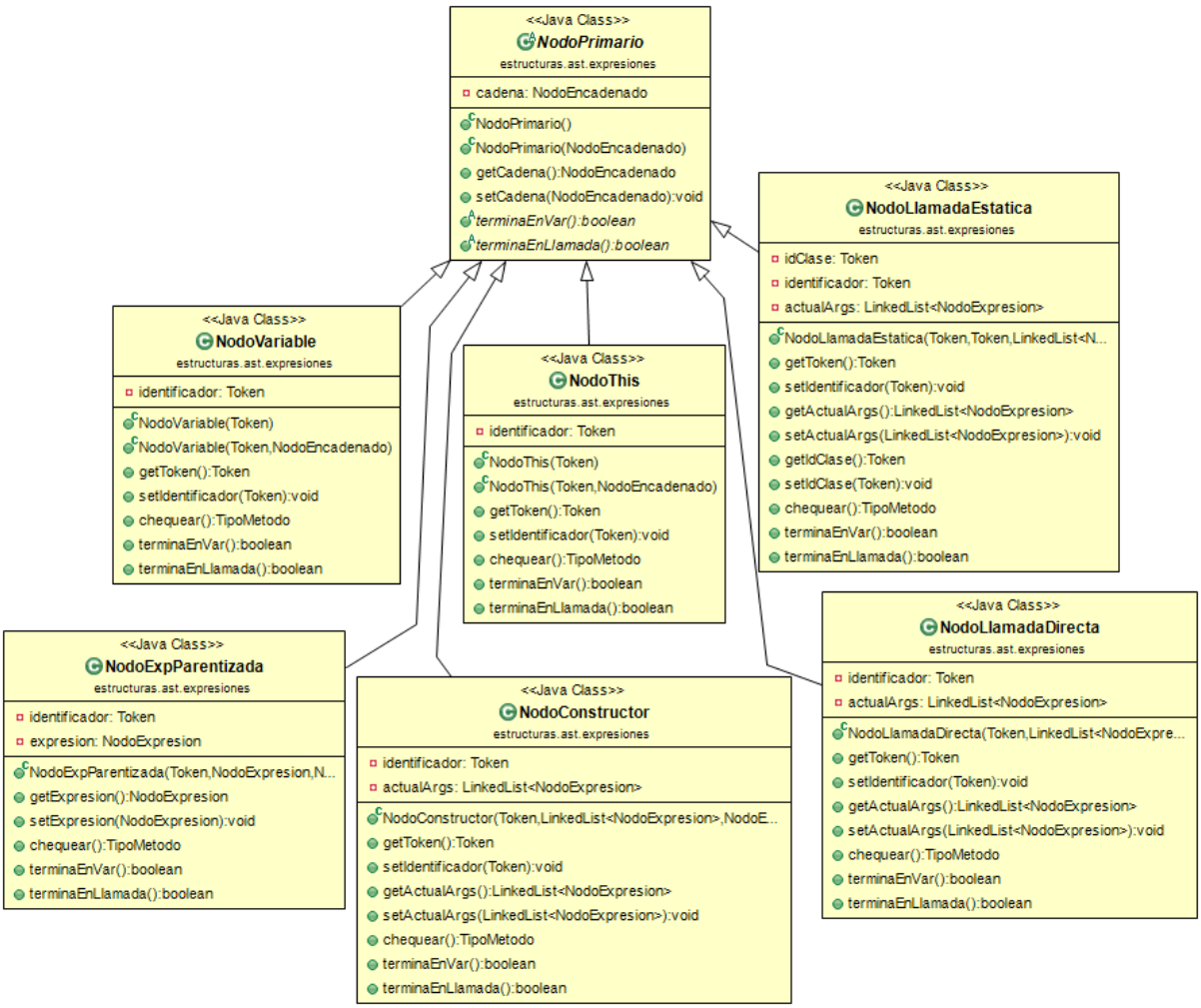

<<Java Class>>
Clase
estructuras.ts
identificador: Token
herencia: String
constructor: Constructor
metodos: HashMap<String,Metodo>
atributos: HashMap<String,VarInstancia>
estaActualizada: boolean
cantAtrib: int
cantMet: int
id: int
Clase(Token,String)
Clase(Token)
agregarAtributo(VarInstancia):void
agregarMetodo(Metodo):void
getConstructor():Constructor
setConstructor(Constructor):void
getMetodos():HashMap<String,Metodo>
getAtributos():HashMap<String,VarInstancia>
estaActualizada():boolean
setEstaActualizada(boolean):void
setPadre(String):void
getNombreClase():String
getNombreClasePadre():String
getToken():Token
chequearDeclaraciones():void
chequearSentencias():void
chequearConstructor():void
actualizar():void
getCantAtrib():int
setCantAtrib(int):void
getCantMet():int
setCantMet(int):void
getLabel():String
getId():int

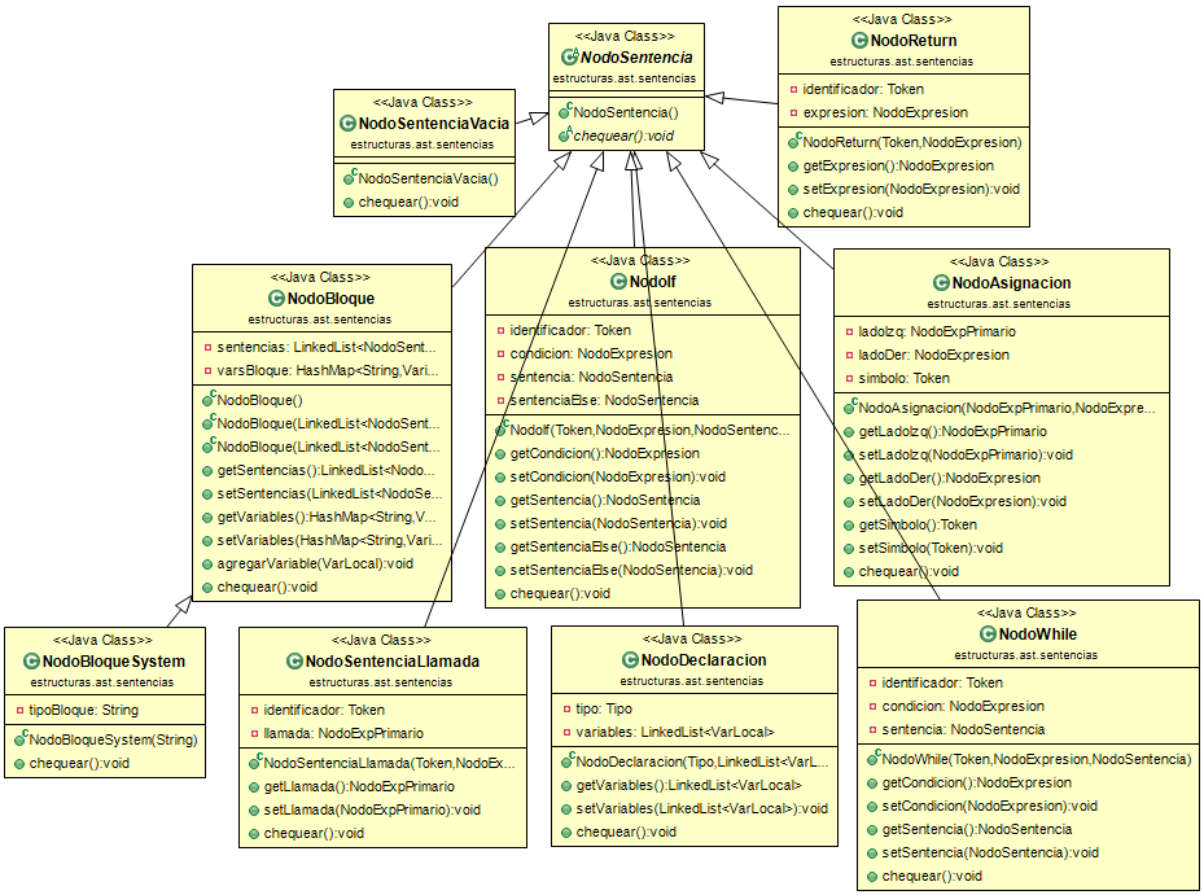

<<Java Class>>
TablaSimbolos
estructuras.ts
claseActual: Clase
metodoActual: Unidad
bloqueActual: NodoBloque
clases: HashMap<String,Clase>
main: Metodo
TablaSimbolos()
agregarClase(String,Clase):void
estaClase(String):boolean
chequearDeclaraciones():void
actualizarMetodos():void
chequearSentencias():void
chequearMain():Metodo
chequearHerenciaCircular():void
chequearHC(String,String):void
chequearAtributos():void
imprimirEstado():void
generarCodigo():void
codigoArranque():void

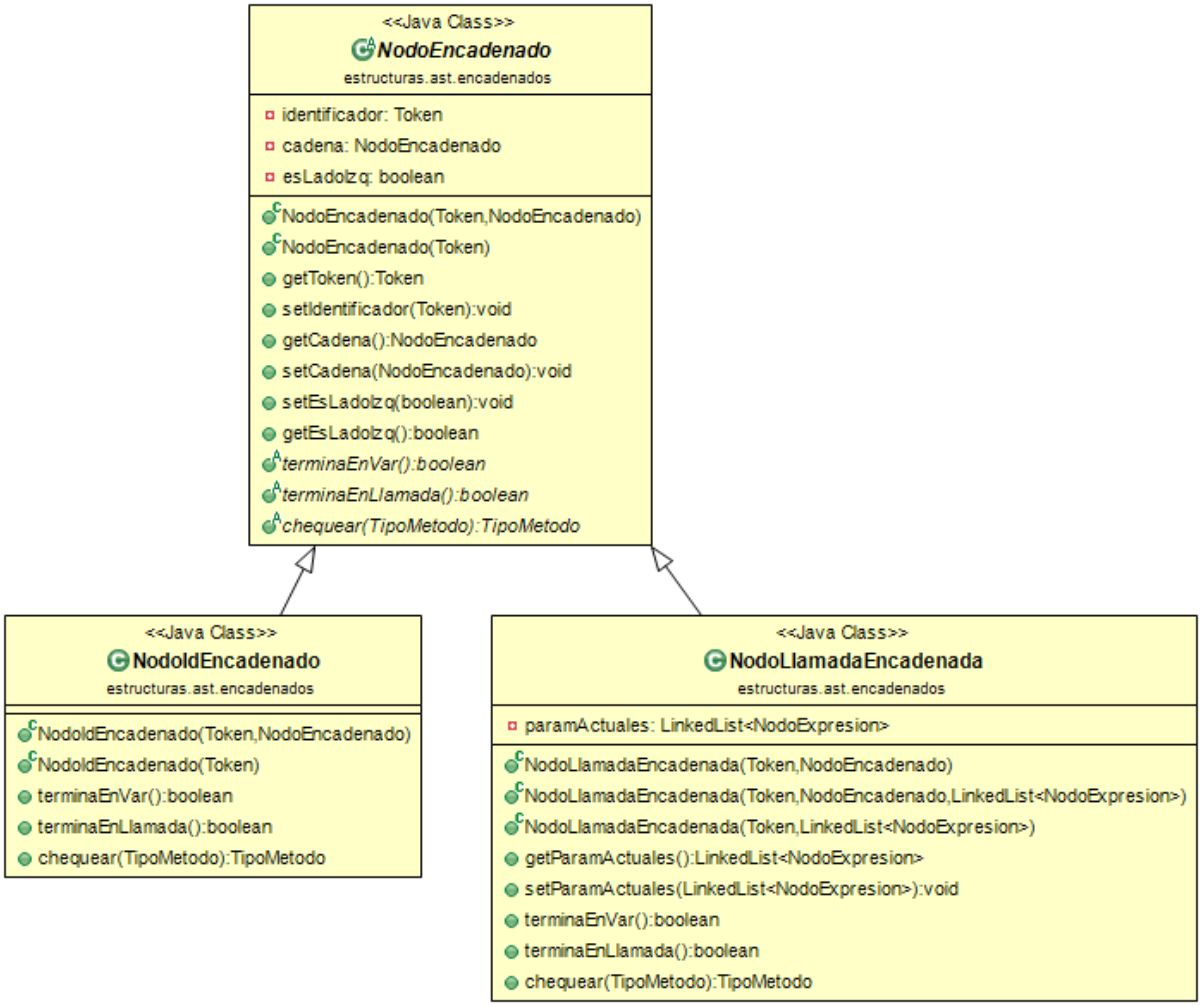

<<Java Class>>
TipoMetodo
estructuras.ts.tipos
TipoMetodo()
getNombreClase():String
esValido():boolean
esVoid():boolean
esTipoPrimitivo():boolean
esTipoClase():boolean
conforma(TipoMetodo):boolean
equals(TipoMetodo):boolean
<<Java Class>>
TipoVoid
estructuras.ts.tipos
TipoVoid()
esValido():boolean
esVoid():boolean
esTipoPrimitivo():boolean
getNombreClase():String
conforma(TipoMetodo):boolean
esTipoClase():boolean
<<Java Class>>
Tipo
estructuras.ts.tipos
Tipo()
esTipoPrimitivo():boolean
getNombreClase():String
esVoid():boolean
<<Java Class>>
TipoPrimitivo
estructuras.ts.tipos
TipoPrimitivo()
esTipoPrimitivo():boolean
esValido():boolean
conforma(TipoMetodo):boolean
esTipoClase():boolean
<<Java Class>>
TipoClase
estructuras.ts.tipos
nombre: Token
TipoClase(Token)
getNombreClase():String
esTipoPrimitivo():boolean
conforma(TipoMetodo):boolean
esValido():boolean
esTipoClase():boolean
<<Java Class>>
TipoChar
estructuras.ts.tipos
TipoChar()
getNombreClase():String
<<Java Class>>
TipoBoolean
estructuras.ts.tipos
TipoBoolean()
getNombreClase():String
<<Java Class>>
TipoInt
estructuras.ts.tipos
TipoInt()
getNombreClase():String
<<Java Class>>
TipoString
estructuras.ts.tipos
TipoString()
getNombreClase():String
<<Java Class>>
TipoNull
estructuras.ts.tipos
TipoNull(Token)
getNombreClase():String
conforma(TipoMetodo):boolean


<<Java Class>>
Variable
estructuras.ts.variables
tipoVar: Tipo
identificador: Token
offset: int
Variable(Token,Tipo)
getTipoVar():Tipo
setTipoVar(Tipo):void
getToken():Token
setIdentificador(Token):void
getOffset():int
setOffset(int):void
compareTo(Variable):int
<<Java Class>>
VarMetodo
estructuras.ts.variables
VarMetodo(Token,Tipo)
<<Java Class>>
VarInstancia
estructuras.ts.variables
esPublico: boolean
VarInstancia(Token,Tipo,boolean)
getEsPublico():boolean
setEsPublico(boolean):void
<<Java Class>>
VarLocal
estructuras.ts.variables
VarLocal(Token,Tipo)
<<Java Class>>
VarParametro
estructuras.ts.variables
VarParametro(Token,Tipo)


<<Java Class>>
NodoExpresion
estructuras.ast.expresiones
NodoExpresion()
chequear():TipoMetodo
<<Java Class>>
NodoExpresionBinaria
estructuras.ast.expresiones
ladoIzq: NodoExpresion
ladoDer: NodoExpresion
simbolo: Token
NodoExpresionBinaria(Token,NodoExpresio...
getLadoIzq():NodoExpresion
setLadoIzq(NodoExpresion):void
getLadoDer():NodoExpresion
setLadoDer(NodoExpresion):void
getSimbolo():Token
setSimbolo(Token):void
chequear():TipoMetodo
<<Java Class>>
NodoExpresionUnaria
estructuras.ast.expresiones
ladoDer: NodoExpresion
simbolo: Token
NodoExpresionUnaria(Token,NodoExpresion)
getLadoDer():NodoExpresion
setLadoDer(NodoExpresion):void
getSimbolo():Token
setSimbolo(Token):void
chequear():TipoMetodo
<<Java Class>>
NodoExpPrimario
estructuras.ast.expresiones
esLadoIzq: boolean
NodoExpPrimario()
setEsLadoIzq(boolean):void
getEsLadoIzq():boolean
<<Java Class>>
NodoLiteral
estructuras.ast.expresiones
literal: Token
NodoLiteral(Token)
getToken():Token
setToken(Token):void
chequear():Tipo
<<Java Class>>
NodoPrimario
estructuras.ast.expresiones
cadena: NodoEncadenado
NodoPrimario()
NodoPrimario(NodoEncadenado)
getCadena():NodoEncadenado
setCadena(NodoEncadenado):void
terminaEnVar():boolean
terminaEnLlamada():boolean


<<Java Class>>
NodoPrimario
estructuras.ast.expresiones
cadena: NodoEncadenado
NodoPrimario()
NodoPrimario(NodoEncadenado)
getCadena():NodoEncadenado
setCadena(NodoEncadenado):void
terminaEnVar():boolean
terminaEnLlamada():boolean
<<Java Class>>
NodoLlamadaEstatica
estructuras.ast.expresiones
idClase: Token
identificador: Token
actualArgs: LinkedList<NodoExpresion>
NodoLlamadaEstatica(Token,Token,LinkedList<N...
getToken():Token
setIdentificador(Token):void
getActualArgs():LinkedList<NodoExpresion>
setActualArgs(LinkedList<NodoExpresion>):void
getIdClase():Token
setIdClase(Token):void
chequear():TipoMetodo
terminaEnVar():boolean
terminaEnLlamada():boolean
<<Java Class>>
NodoVariable
estructuras.ast.expresiones
identificador: Token
NodoVariable(Token)
NodoVariable(Token,NodoEncadenado)
getToken():Token
setIdentificador(Token):void
chequear():TipoMetodo
terminaEnVar():boolean
terminaEnLlamada():boolean
<<Java Class>>
NodoThis
estructuras.ast.expresiones
identificador: Token
NodoThis(Token)
NodoThis(Token,NodoEncadenado)
getToken():Token
setIdentificador(Token):void
chequear():TipoMetodo
terminaEnVar():boolean
terminaEnLlamada():boolean
<<Java Class>>
NodoExpParentizada
estructuras.ast.expresiones
identificador: Token
expresion: NodoExpresion
NodoExpParentizada(Token,NodoExpresion,N...
getExpresion():NodoExpresion
setExpresion(NodoExpresion):void
chequear():TipoMetodo
terminaEnVar():boolean
terminaEnLlamada():boolean
<<Java Class>>
NodoConstructor
estructuras.ast.expresiones
identificador: Token
actualArgs: LinkedList<NodoExpresion>
NodoConstructor(Token,LinkedList<NodoExpresion>,NodoE...
getToken():Token
setIdentificador(Token):void
getActualArgs():LinkedList<NodoExpresion>
setActualArgs(LinkedList<NodoExpresion>):void
chequear():TipoMetodo
terminaEnVar():boolean
terminaEnLlamada():boolean
<<Java Class>>
NodoLlamadaDirecta
estructuras.ast.expresiones
identificador: Token
actualArgs: LinkedList<NodoExpresion>
NodoLlamadaDirecta(Token,LinkedList<NodoExpre...
getToken():Token
setIdentificador(Token):void
getActualArgs():LinkedList<NodoExpresion>
setActualArgs(LinkedList<NodoExpresion>):void
chequear():TipoMetodo
terminaEnVar():boolean
terminaEnLlamada():boolean


<<Java Class>>
NodoSentencia
estructuras.ast.sentencias
NodoSentencia()
chequear():void
<<Java Class>>
NodoReturn
estructuras.ast.sentencias
identificador: Token
expresion: NodoExpresion
NodoReturn(Token,NodoExpresion)
getExpresion():NodoExpresion
setExpresion(NodoExpresion):void
chequear():void
<<Java Class>>
NodoSentenciaVacia
estructuras.ast.sentencias
NodoSentenciaVacia()
chequear():void
<<Java Class>>
NodoBloque
estructuras.ast.sentencias
sentencias: LinkedList<NodoSent...
varsBloque: HashMap<String,Vari...
NodoBloque()
NodoBloque(LinkedList<NodoSent...
NodoBloque(LinkedList<NodoSent...
getSentencias():LinkedList<Nodo...
setSentencias(LinkedList<NodoSe...
getVariables():HashMap<String,V...
setVariables(HashMap<String,Vari...
agregarVariable(VarLocal):void
chequear():void
<<Java Class>>
NodoIf
estructuras.ast.sentencias
identificador: Token
condicion: NodoExpresion
sentencia: NodoSentencia
sentenciaElse: NodoSentencia
NodoIf(Token,NodoExpresion,NodoSentenc...
getCondicion():NodoExpresion
setCondicion(NodoExpresion):void
getSentencia():NodoSentencia
setSentencia(NodoSentencia):void
getSentenciaElse():NodoSentencia
setSentenciaElse(NodoSentencia):void
chequear():void
<<Java Class>>
NodoAsignacion
estructuras.ast.sentencias
ladoIzq: NodoExpPrimario
ladoDer: NodoExpresion
simbolo: Token
NodoAsignacion(NodoExpPrimario,NodoExpre...
getLadoIzq():NodoExpPrimario
setLadoIzq(NodoExpPrimario):void
getLadoDer():NodoExpresion
setLadoDer(NodoExpresion):void
getSimbolo():Token
setSimbolo(Token):void
chequear():void
<<Java Class>>
NodoBloqueSystem
estructuras.ast.sentencias
tipoBloque: String
NodoBloqueSystem(String)
chequear():void
<<Java Class>>
NodoSentenciaLlamada
estructuras.ast.sentencias
identificador: Token
llamada: NodoExpPrimario
NodoSentenciaLlamada(Token,NodoEx...
getLlamada():NodoExpPrimario
setLlamada(NodoExpPrimario):void
chequear():void
<<Java Class>>
NodoDeclaracion
estructuras.ast.sentencias
tipo: Tipo
variables: LinkedList<VarLocal>
NodoDeclaracion(Tipo,LinkedList<VarL...
getVariables():LinkedList<VarLocal>
setVariables(LinkedList<VarLocal>):void
chequear():void
<<Java Class>>
NodoWhile
estructuras.ast.sentencias
identificador: Token
condicion: NodoExpresion
sentencia: NodoSentencia
NodoWhile(Token,NodoExpresion,NodoSentencia)
getCondicion():NodoExpresion
setCondicion(NodoExpresion):void
getSentencia():NodoSentencia
setSentencia(NodoSentencia):void
chequear():void


<<Java Class>>
NodoEncadenado
estructuras.ast.encadenados
identificador: Token
cadena: NodoEncadenado
esLadoIzq: boolean
NodoEncadenado(Token,NodoEncadenado)
NodoEncadenado(Token)
getToken():Token
setIdentificador(Token):void
getCadena():NodoEncadenado
setCadena(NodoEncadenado):void
setEsLadoIzq(boolean):void
getEsLadoIzq():boolean
terminaEnVar():boolean
terminaEnLlamada():boolean
chequear(TipoMetodo):TipoMetodo
<<Java Class>>
NodoIdEncadenado
estructuras.ast.encadenados
NodoIdEncadenado(Token,NodoEncadenado)
NodoIdEncadenado(Token)
terminaEnVar():boolean
terminaEnLlamada():boolean
chequear(TipoMetodo):TipoMetodo
<<Java Class>>
NodoLlamadaEncadenada
estructuras.ast.encadenados
paramActuales: LinkedList<NodoExpresion>
NodoLlamadaEncadenada(Token,NodoEncadenado)
NodoLlamadaEncadenada(Token,NodoEncadenado,LinkedList<NodoExpresion>)
NodoLlamadaEncadenada(Token,LinkedList<NodoExpresion>)
getParamActuales():LinkedList<NodoExpresion>
setParamActuales(LinkedList<NodoExpresion>):void
terminaEnVar():boolean
terminaEnLlamada():boolean
chequear(TipoMetodo):TipoMetodo

5.4. Esquema de Traducción

A continuación se presenta el Esquema de Traducción (EDT) de la gramática.

<*Inicial*> → (1) <*Clase*><*InicialPrimo*>

```
(1) (Por única vez)
TS = new TablaSimbolos()
TS.agregarClase(Object)
TS.agregarClase(System)
```

<*InicialPrimo*> → <*Inicial*> <*InicialPrimo*> → λ
<*Clase*> → `class idClase` **(1)** <*Herencia*> { <*ListaMiembro*> } **(2)**

```
(1) TS.agregarClase(idClase)
claseActual = idClase
(2) Si no tiene constructor agrego uno por defecto.
```

<*Herencia*> → `extends idClase` **(1)**

```
(1) claseActual.setPadre(idClase)
```

<*Herencia*> → λ **(1)**

```
(1) claseActual.setPadre("Object")
```

<*ListaMiembro*> → <*Miembro*> <*ListaMiembro*>

<*ListaMiembro*> → λ

<*Miembro*> → <*Atributo*>

<*Miembro*> → <*Ctor*>

<*Miembro*> → <*Metodo*>

<*Atributo*> → <*Visibilidad*> **(1)** <*Tipo*> **(2)** <*ListaDecVars*> **(3)(4)** ;

```
(1) Atributo.esPublico=Visibilidad.esPublico
(2) Atributo.tipo=Tipo.tipo
(3) Atributo.atributos=ListaDecVars.atributos
(4) for each (Atributo.atributos)
vi = new VarInstancia(Atributo.visibilidad, Atributo.tipo)
claseActual.add(vi)
```

<*Metodo*> → <*FormaMetodo*> **(1)** <*TipoMetodo*> **(2)** `idMetVar` <*ArgsFormales*> **(3)** <*Bloque*> **(4)**

```
(1) Metodo.esEstatico = FormaMetodo.esEstatico
(2) Metodo.tipo = TipoMetodo.tipo
(3) met = new Metodo(idMetVar, Metodo.esEstatico Metodo.tipo)
claseActual.addMetodo(met)
metodoActual = met
(4) Metodo.bloque = Bloque.bloque
metodoActual = setCodigo(Metodo.bloque)
```

<*Ctor*> → `idClase` **(1)** <*ArgsFormales*> <*Bloque*> **(2)**

```
(1) Ctor = new Constructor(idClase)
metodoActual = Ctor
(2) Ctor.bloque = Bloque.bloque
metodoActual.setBloque(Ctor.bloque)
```

<*Visibilidad*> → `public` **(1)**

```
(1) Visibilidad.esPublico = true
```

<*Visibilidad*> → `private` **(1)**

```
(1) Visibilidad.esPublico = false
```

<*ArgsFormales*> → (<*LAForm*>

<*LAForm*> → <*ListaArgsFormales*>)

<*LAForm*> →)

<*ListaArgsFormales*> → <*ArgFormal*> <*RestoLAF*>

<*RestoLAF*> → , <*ListaArgsFormales*>

<*RestoLAF*> → λ

<*ArgFormal*> → <*Tipo*> `idMetVar` **(1)**

```
(1) vp = new VarParametro(idMetVar, Tipo.tipo)
metodoActual.agregarParametro(var)
```

<*FormaMetodo*> → `static` **(1)**

```
(1) FormaMetodo.esEstatico = true
```

<*FormaMetodo*> → `dynamic` **(1)**

```
(1) FormaMetodo.esEstatico = false
```

<*TipoMetodo*> → <*Tipo*> **(1)**

```
(1) TipoMetodo.tipo = Tipo.tipo
```

<*TipoMetodo*> → `void` **(1)**

```
(1) TipoMetodo.tipo = new TipoVoid()
```

<*Tipo*> → <*TipoPrimitivo*> **(1)**

```
(1) Tipo.tipo = TipoPrimitivo.tipo
```

<*Tipo*> → `idClase` **(1)**

```
(1) Tipo.tipo = new TipoClase(idClase)
```

<*TipoPrimitivo*> → `boolean` **(1)**

```
(1) TipoPrimitivo.tipo = new TipoBoolean()
```

<*TipoPrimitivo*> → `char` **(1)**

```
(1) TipoPrimitivo.tipo = new TipoChar()
```

<*TipoPrimitivo*> → `int` **(1)**

```
(1) TipoPrimitivo.tipo = new TipoInt()
```

<*TipoPrimitivo*> → `String` **(1)**

```
(1) TipoPrimitivo.tipo = new TipoString()
```

<*ListaDecVars*> → `idMetVar` **(1)** <*RestoLDV*> **(2)(3)**

```
(1) var = new Token(idMetVar)
(2) ListaDecVars.lista = RestoLDV.lista
(3) ListaDecVars.lista.addFirst(var)
```

<*RestoLDV*> → , <*ListaDecVars*> **(1)**

```
(1) RestoLDV.lista = ListaDecVars.lista
```

<*RestoLDV*> → λ **(1)**

```
(1) RestoLDV.lista = new Lista()
```

<*Bloque*> → { <*ListaSentencias*> **(1)** }

```
(1) Bloque.bloque = new NodoBloque(ListaSentencias.lista)
```

<*ListaSentencias*> → <*Sentencia*> <*ListaSentencias1*> **(1)**

```
(1) sen = new NodoSentencia(Sentencia.sen)
ListaSentencias.lista = ListaSentencias1.lista
ListaSentencias.lista.addFirst(sen)
```

<*ListaSentencias*> → λ **(1)**

```
(1) ListaSentencias.lista = new Lista()
```

<*Sentencia*> → ; **(1)**

```
(1) Sentencia.sent = new NodoSentenciaVacia()
```

<*Sentencia*> → <*Primario*> <*AsignacionOSentenciaLlamada*> ; **(1)**

```
(1) if (AsignacionOSentenciaLlamada.expr = null)
Sentencia.sent = new NodoSentenciaLlamada(Primario.sent)
else
expr = AsignacionOSentenciaLlamada.expr
Sentencia.sent = new NodoAsginacion(Primario.sent, expr)
```

<*Sentencia*> → <*Tipo*> **(1)** <*ListaDecVars*> **(2)(3)**

```
(1) Sentencia.tipo = Tipo.tipo
(2) L = new Lista()
(3) for each (elemento de ListaDecVars.lista)
vl = new VarLocal (Sentencia.tipo)
L.add(vl)
Sentencia.sent = new NodoDeclaracion(Sentencia.tipo, L)
```

<*Sentencia*> → `if (` <*Expresion*> `)` <*Sentencia1*> <*ElseOpc*> **(1)**

```
(1) Sentencia.sent = new NodoIf(Expresion.expr, Sentencia1.sent, ElseOpc.sent)
```

<*ElseOpc*> → `else` <*Sentencia*> **(1)**

```
(1) ElseOpc.sent = Sentencia.sent
```

<*ElseOpc*> → λ **(1)**

```
(1) ElseOpc.sent = null
```

<*Sentencia*> → `while (` <*Expresion*> `)` <*Sentencia1*> **(1)**

```
(1) Sentencia.sent = new NodoWhile(Expresion.expr, Sentencia1.sent)
```

<*Sentencia*> → <*Bloque*> **(1)**

```
(1) Sentencia.sent = Bloque
```

<*Sentencia*> → `return` <*ExpresionPrimo*> **(1)**

```
(1) Sentencia.sent = new NodoReturn(ExpresionPrimo.expr)
```

<*ExpresionPrimo*> → <*Expresion*> ; **(1)**

```
(1) ExpresionPrimo.expr = Expresion.expr
```

<*ExpresionPrimo*> → ; **(1)**

```
(1) ExpresionPrimo.expr = null
```

<*AsignacionOSentenciaLlamada*> → = <*Expresion*> **(1)**

```
(1) AsignacionOSentenciaLlamada.expr = Expresion.expr
```

<*AsignacionOSentenciaLlamada*> → λ **(1)**

```
(1) AsignacionOSentenciaLlamada.sent = null
```

<*Expresion*> → <*ExpOr*> **(1)**

```
(1) Expresion.expr = ExpOr.expr
```

<*ExpOr*> → <*ExpAnd*> **(1)** <*ExpOrPrimo*> **(2)**

```
(1) ExpOr.expr1 = ExpAnd.expr
    ExpOrPrimo.expr1 = ExpOr.expr1
(2) ExpOr.expr = ExpOrPrimo.expr
```

<*ExpOrPrimo*> → || <*ExpAnd*> **(1)** <*ExpOrPrimo1*> **(2)**

```
(1) ExpOrPrimo.expr2 = ExpAnd.expr
ExpOrPrimo.expr3 = new NodoExpresionBinaria(expr1, expr2, OR)
ExpOrPrimo1.expr1 = ExpOrPrimo.expr3
(2) ExpOrPrimo.expr = ExpOrPrimo1.expr
```

<*ExpOrPrimo*> → λ **(1)**

```
(1) ExpOrPrimo.expr = ExpOrPrimo.expr1
```

<*ExpAnd*> → <*ExpIg*> **(1)** <*ExpAndPrimo*> **(2)**

```
(1) ExpAnd.expr1 = ExpIg.expr
    ExpAndPrimo.expr1 = ExpAnd.expr1
(2) ExpAnd.expr = ExpAndPrimo.expr
```

<*ExpAndPrimo*> → && <*ExpIg*> **(1)** <*ExpAndPrimo1*> **(2)**

```
(1) ExpAndPrimo.expr2 = ExpIg.expr
ExpAndPrimo.expr3 = new NodoExpresionBinaria(expr1, expr2, AND)
ExpAndPrimo1.expr1 = ExpAndPrimo.expr3
(2) ExpAndPrimo.expr = ExpAndPrimo1.expr
```

<*ExpAndPrimo*> → λ **(1)**

```
(1) ExpAndPrimo.expr = ExpAndPrimo.expr1
```

<*ExpIg*> → <*ExpComp*> **(1)** <*ExpIgPrimo*> **(2)**

```
(1) ExpIg.expr1 = ExpComp.expr
ExpIgPrimo.expr1 = ExpIg.expr1
(2) ExpIg.expr = ExpIgPrimo.expr
```

<*ExpIgPrimo*> → <*OpIg*> <*ExpComp*> **(1)** <*ExpIgPrimo1*> **(2)**

```
(1) ExpIgPrimo.expr2 = ExpComp.expr
ExpIgPrimo.op = OpIg.op
ExpIgPrimo.expr3 = new NodoExpresionBinaria(expr1, expr2, ExpIgPrimo.op)
ExpIgPrimo1.expr1 = ExpIgPrimo.expr3
(2) ExpIgPrimo.expr = ExpIgPrimo1.expr
```

<*ExpIgPrimo*> → λ **(1)**

```
(1) ExpIgPrimo.expr = ExpIgPrimo.expr1
```

<*ExpComp*> → <*ExpAd*> **(1)** <*RestoExpComp*> **(2)**

```
(1) ExpComp.expr1 = ExpAd.expr
RestoExpComp.expr1 = ExpComp.expr1
(2) ExpComp.expr = RestoExpComp.expr
```

<*RestoExpComp*> → <*OpComp*> <*ExpAd*> **(1)**

```
(1) RestoExpComp.op = OpComp.op
RestoExpComp.expr2 = ExpAd.expr
RestoExpComp.expr = new NodoExpresionBinaria(expr1, expr2, RestoExpComp.op)
```

<*RestoExpComp*> → `instanceof idClase` **(1)**

```
(1) RestoExpComp.expr =  new NodoExpresionBinaria(expr1, idClase, instanceof)
```

<*RestoExpComp*> → λ **(1)**

```
(1) RestoExpComp.expr = RestoExpComp.expr1
```

<*ExpAd*> → <*ExpMul*> **(1)** <*ExpAdPrimo*> **(2)**

```
(1) ExpAd.expr1 = ExpMul.expr
ExpAdPrimo.expr1 = ExpAd.expr1
(2) ExpAd.expr = ExpAdPrimo.expr
```

<*ExpAdPrimo*> → <*OpAd*> <*ExpMul*> **(1)** <*ExpAdPrimo1*> **(2)**

```
(1) ExpAdPrimo.op = OpAd.op
ExpAdPrimo.expr2 = ExpMul.expr
ExpAdPrimo.expr3 = new NodoExpresionBinaria(expr1, expr2, ExpAdPrimo.op)
ExpAdPrimo1.expr1 = ExpAdPrimo.expr
(2) ExpAdPrimo.expr = ExAdPrimo1.expr
```

<*ExpAdPrimo*> → λ **(1)**

```
(1) ExpAdPrimo.expr = ExpAdPrimo.expr1
```

<*ExpMul*> → <*ExpUn*> **(1)** <*ExpMulPrimo*> **(2)**

```
(1) ExpMul.expr1 = ExpUn.expr
ExpMulPrimo.expr1 = ExpMul.expr1
(2) ExpMul.expr = ExpMulPrimo.expr
```

<*ExpMulPrimo*> → <*OpMul*> <*ExpUn*> **(1)** <*ExpMulPrimo1*> **(2)**

```
(1) ExpMulPrimo.op = OpMul.op
ExpMulPrimo.expr2 = ExpUn.expr
ExpMulPrimo.expr3 = new NodoExpresionBinaria(expr1, expr2, OpMul.op)
ExpMulPrimo1.expr1 = ExpMulPrimo.expr3
(2) ExpMulPrimo.expr = ExpMulPrimo1.expr
```

<*ExpMulPrimo*> → λ **(1)**

```
(1) ExpMulPrimo.expr = ExpMulPrimo.expr1
```

<*ExpUn*> → <*OpUn*> <*ExpUn1*> **(1)**

```
(1) ExpUn.op = OpUn.op
ExpUn.expr1 = ExpUn1.expr
ExpUn.expr = new NodoExpresionUnaria(ExpUn.op,ExpUn.expr1)
```

<*ExpUn*> → <*ExpCast*> **(1)**

```
(1) ExpUn.expr = ExpCast.expr
```

<*ExpCast*> → `[ idClase ]`<*Operando*> **(1)**

```
(1) ExpCast.expr1 = Operando.expr
ExpCast.expr = new NodoExpresionUnaria(idClase, ExpCast.expr1)
```

<*ExpCast*> → <*Operando*> **(1)**

```
(1) ExpCast.expr = Operando.expr
```

<*OpIg*> → == **(1)**

```
(1) OpIg.op = ''==''
```

<*OpIg*> → != **(1)**

```
(1) OpIg.op = ''!=''
```

<*OpComp*> → > **(1)**

```
(1) OpComp.op = ''>''
```

<*OpComp*> → < **(1)**

```
(1) OpComp.op = ''<''
```

<*OpComp*> → >= **(1)**

```
(1) OpComp.op = ''>=''
```

<*OpComp*> → <= **(1)**

```
(1) OpComp.op = ''<=''
```

<*OpAd*> → + **(1)**

```
(1) OpAd.op = ''+''
```

<*OpAd*> → - **(1)**

```
(1) OpAd.op = ''-''
```

<*OpUn*> → + **(1)**

```
(1) OpUn.op = ''+''
```

<*OpUn*> → - **(1)**

```
(1) OpUn.op = ''-''
```

<*OpUn*> → ! **(1)**

```
(1) OpUn.op = ''!''
```

<*OpMul*> → * **(1)**

```
(1) OpMul.op = ''*''
```

<*OpMul*> → / **(1)**

```
(1) OpMul.op = ''/''
```

<*OpMul*> → % **(1)**

```
(1) OpMul.op = ''%''
```

<*Operando*> → <*Literal*> **(1)**

```
(1) Operando.exp = Literal.exp
```

<*Operando*> → <*Primario*> **(1)**

```
(1) Operando.exp = Primario.exp
```

<*Literal*> → `null` **(1)**

```
(1) Literal.exp = new NodoLiteral(''null'')
```

<*Literal*> → `true` **(1)**

```
(1) Literal.exp = new NodoLiteral(''true'')
```

<*Literal*> → `false` **(1)**

```
(1) Literal.exp = new NodoLiteral(''false'')
```

<*Literal*> → `intLiteral` **(1)**

```
(1) Literal.exp = new NodoLiteral(intLiteral.lexema)
```

<*Literal*> → `charLiteral` **(1)**

```
(1) Literal.exp = new Literal(charLiteral.lexema)
```

<*Literal*> → `stringLiteral` **(1)**

```
(1) Literal.exp = new NodoLiteral(stringLiteral.lexema)
```

<*Primario*> → (<*PrimarioUno*> **(1)**

```
(1) Primario.expr = PrimarioUno.expr
```

<*PrimarioUno*> → <*Expresion*>) <*ListaLOI*> **(1)**

```
(1) nodo = new NodoExpParentizada(Expresion.expr)
nodo.setEncadenado(ListaLOI.lista)
PrimarioUno.expr = nodo
```

<*PrimarioUno*> → `idClase . idMetVar` <*ArgsActuales*>) <*ListaLOI*> **(1)**

```
(1) lista = ArgsActuales.lista
PrimarioUno.exp = new NodoLlamadaEstatica(idClase,idMetVar,lista)
```

<*Primario*> → `this` <*ListaLOI*> **(1)**

```
(1) Primario.expr = new NodoThis()
Primario.expr.setCadena(ListaLOI.lista)
```

<*Primario*> → `idMetVar` **(1)** <*PrimarioDos*> **(2)**

```
(1) PrimarioDos.expr1 = idMetVar
(2) Primario.expr = PrimarioDos.expr
```

<*Primario*> → `new idClase` <*ArgsActuales*> <*ListaLOI*> **(1)**

```
(1) lista = ArgsActuales.lista
Primario.expr = new NodoConstructor(idClase,lista)
Primario.expr.setCadena(ListaLOI.lista)
```

<*PrimarioDos*> → <*ListaLOI*> **(1)**

```
(1) PrimarioDos.expr = new NodoVariable(expr1, ListaLOI.lista)
```

<*PrimarioDos*> → <*ArgsActuales*> <*ListaLOI*> **(1)**

```
(1) PrimarioDos.expr = new NodoLlamadaDirecta(expr1, ArgsActuales.lista)
PrimarioDos.expr.setCadena(ListaLOI.lista)
```

<*ListaLOI*> → <*LlamadaoIdEncadenado*> <*ListaLOI1*> **(1)**

```
(1) ListaLOI.lista = LlamadaoIdEncadenado.lista
ListaLOI.setCadena(ListaLOI1.lista)
```

<*ListaLOI*> → λ **(1)**

```
(1) ListaLOI.lista = null
```

<*LlamadaoIdEncadenado*> → `. idMetVar` <*RestoLOI*> **(1)**

```
(1) lista = RestoLOI.lista
if (lista=null)
then LlamadaoIdEncadenado.lista = new NodoIdEncadenado(expr)
else LlamadaoIdEncadenado.lista = new NodoLlamadaEncadenada(expr,lista)
```

<*RestoLOI*> → <*ArgsActuales*> **(1)**

```
(1) RestoLOI.lista = ArgsActuales.lista
```

<*RestoLOI*> → λ **(1)**

```
(1) RestoLOI.lista = null
```

<*ArgsActuales*> → (<*ListaExpsOpc*>) **(1)**

```
(1) ArgsActuales.lista = ListaExpsOpc.lista
```

<*ListaExpsOpc*> → <*ListaExps*> **(1)**

```
(1) ListaExpsOpc.lista = ListaExps.lista
```

<*ListaExpsOpc*> → λ **(1)**

```
(1) ListaExpsOpc.lista = new Lista()
```

<*ListaExps*> → <*Expresion*> <*RestoListaExps*> **(1)**

```
(1) ListaExps.add = Expresion.exp
ListaExps.addLast(RestoListaExps.lista)
```

<*RestoListaExps*> → , <*ListaExps*> **(1)**

```
(1) RestoListaExps.lista = ListaExps.addLast(ListaExps.lista)
```

<*RestoListaExps*> → λ **(1)**

```
(1) RestoListaExps.lista = new Lista()
```

5.5. Decisiones sobre el diseño

- `TipoNull` hereda de `TipoClase` con el fin de facilitar la conformidad de tipos. Recordar que se puede asignar `null` a cualquier objeto de tipo clase.
- El tipo `String` es representado como un tipo primitivo, y para facilidad de implementación se supone que no se le puse asignar un valor `null` a algo de tipo `String`. Esto no sucede en JAVA.
- Algunos nodos de la estructura del AST guardan una referencia al token en cuestión, con la única finalidad de tener guardado el número de línea para mostrar un error, si así sucediera. Esto pasa por ejemplo con los casos del `NodoIf` y `NodoWhile`, que se pueden utilizar sin necesidad de guardar el token. Sin embargo, sin el número de línea, no se podría dar un mensaje de error preciso.
- Por simplicidad, el operador de casteo se representó como si fuera una expresión unaria, mientras que el `instanceof` como si fuera una expresión binaria. Se puede modificar la estructura del AST creando dos nuevas clases específicas para esto, y además modificar el analizador sintáctico para que cree los nodos correspondientes. Sin embargo, aprovechando el uso de la jerarquía realizada para los AST, resultaba más simple representar estas dos estructuras mediante dichos nodos.

5.6. Pruebas utilizadas

Siguiendo con la costumbre de la batería de pruebas anteriores, para la prueba del analizador semántico se crearon dos conjuntos de pruebas: los de clase A (correctos) y los de clase B (incorrectos).

Para las pruebas de clase A se crearon programas sintáctica y semánticamente válidos, cuyo objetivo era probar el correcto funcionamiento de los métodos estáticos, métodos dinámicos, atributos que no eran visibles públicamente por clases que heredaban dichos atributos, variables locales tapando nombre de otras variables, declaracion de variables dentro de bloques anidados, sentencias `if` y `while` anidadas con sus respectivas condiciones booleanas, variables dentro de las expresiones, uso correcto del `instanceof`, distintos tipos de conformidades (con `null` y `Object`, por ejemplo), sentencias de llamada, sentencias de retorno y llamadas con expresiones punto. La batería de pruebas fue pensada en forma incremental sobre los casos más simples (empezando desde la comprobación, herencia y/o redefinición de variables) hasta casos más complejos como pueden ser métodos estáticos llamando a métodos dinámicos o la llamada de métodos heredados.

Para las pruebas de clase B la batería funciona de forma totalmente inversa. Las primeras pruebas realizadas intentaban ver errores semánticos básicos que no debían ser permitidos (como por ejemplo definir una clase con nombre `Object` o `System`), o consideraban casos de errores semánticos típicos, como es el caso de la herencia circular o la herencia de sí misma. También se probó que se esté verificando bien que existe al menos una clase con método estatico `main` cuyo tipo de retorno sea void y no tenga parámetros. Luego se procedieron a hacer pruebas sobre las declaraciones, principalmente utilizando nombres duplicados tanto en clases, métodos como en variables. Por último, las pruebas fueron tomando casos más serios o complejos, como por ejemplo usar una expresión no booleana en la condición de un `if` o `while`, redefiniciones incorrectas sobre métodos heredados, mal uso del `this` (en un método estático), llamada a un método dinámico desde un método estático, y por último varias pruebas sobre conformidad (o disconformidad) de tipos.

6. Generador de Código Intermedio

6.1. Clases Implementadas

6.1.1. Estructuras

Para esta etapa se hizo modificación de varias estructuras del AST y de la tabla de símbolos. Ahora dichas estructuras además de verificar la correctitud semántica del programa, también van generando fragmentos de código que servirán luego para ejecutar en la máquina virtual.

Por otro lado, es necesario realizar la aclaración de que algunas estructuras como por ejemplo los métodos o las variables reciben un nuevo atributo `offset` junto con sus respectivos *setters* y *getters*.

Por último, el cambio que es necesario remarcar es la agregación de una variable `esLadoIzquierdo` a `NodoExpPrimario` y a `NodoEncadenado` que sirve para notificar a las expresiones o sentencias encadenadas que están a la izquierda de una asignación.

NodoBloqueSystem Se agregó esta clase que es una especialización de `NodoBloque`. La clase se encarga de establecer el código (y luego generarlo, cuando se haga el chequeo de sentencias) correspondientes a los métodos que brinda la clase `System`. Es decir, al momento de crear los métodos de `System`, en vez de asociarle un bloque vacío, se le asociará un bloque de este tipo.

6.1.2. Modulos

GenCod Si bien parece ser un módulo, el generador de código intermedio es simplemente una interfaz que ayuda con la generación de etiquetas (labels) durante la generación de código, y por otro lado provee facilidades para escribir en el archivo de salida (es decir, para generar código). Sin embargo, la lógica de la generación de código está internamente en los nodos del AST y algunas estructuras de la tabla de símbolos.

6.2. Estructura del Generador de Código Intermedio

Como se ha mencionado recién, la generación de código intermedio no la hace el módulo de código intermedio, sino que la hacen las estructuras del AST y tabla de símbolos al momento de realizar los controles semánticos.

Dado que la máquina virtual CeIVM opera con una arquitectura de pila, la traducción al código intermedio se hace más sencilla ante la naturaleza recursiva del compilador. Por ejemplo, para un `NodoExpresionBinaria` es necesario generar primero el código de sus dos expresiones correspondientes (que se supone que dejarán en el tope de la pila los valores), para luego generar el código de la operación binaria correspondiente.

6.3. Solución a problemas frecuentes

- **Métodos chequeados varias veces**:
  Para evitar que un método heredado realice el chequeo de sentencias en múltiples ocasiones, a la clase `Metodo` se le agregó el atributo de instancia *ClaseImplementadora* para guardar quien fue la clase que lo definió. Entonces, supongamos que tenemos el método `met1` definido en una clase A, y una clase B que hereda de A, se espera que `met1` realice su chequeo de sentencias solo en A y no en B. Por otro lado, si existiera una clase C que hereda de B, y además implementa a `met1`, este nuevo `met1` tendrá como *ClaseImplementadora* a la clase C, causando que el chequeo se haga solamente para A y para C.

- **Múltiples apariciones del método `main`**:
  Dado que existe la posibilidad de que varias clases implementen el método estático `main`, cuyo retorno sea `void` y no tenga parámetros, ante este caso es necesario tomar un política para decidir cuál será el `main` principal. El motivo es que es necesario saber durante la compilación cuál es el método, para poder obtener su etiqueta y realizar el `CALL` al método que dé inicio al programa. Para poder solucionar este problema, cuando se hacen los controles de declaraciones durante el chequeo semántico, se almacena la cantidad de métodos `main` (estáticos, retorno `void` y sin argumentos) y se guarda una referencia al primero de los encontrados.
  En caso de no existir métodos `main`, se reportará un error. En caso de haber más de un método `main`, se le advertirá al usuario de dicho problema y se informará cuál es el main a considerar, siguiendo con el transcurso de compilación. Si hubiera un único método main, no se reportará nada y el compilador seguirá ejecutando normalmente.
  La política de selección del `main` dependerá de JAVA, ya que el algoritmo tomará el primero que encuentre, dependiendo de cómo esté asociado el conjunto de Clases en el *HashMap* perteneciente a las clases, ubicado en la Tabla de Símbolos (revisar método `chequearMain` en la clase `TablaSimbolos.java`).

- **Métodos con nombre similar**:
  MINIJAVA, al igual que JAVA, es sensible a las mayúsculas, lo que significa que para ambos lenguajes los identificadores "*metodoA*" y "*metodoa*" corresponden a dos métodos diferentes (notar que la diferencia reside en la última letra, cuya A es mayúscula para uno y minúscula para el otro). Lo mismo sucede con los nombres de clase (Por ejemplo "*ClaseA*" y "*Clasea*" corresponden a dos clases diferentes y es sintáctica y semánticamente válido).
  El problema aparece con la generación de etiquetas para la CEIVM, la cual no es sensible a mayúsculas. Una etiqueta en CEIASM es indiferente en cuanto a las mayúsculas y minúsculas. Esto es, "*ClaseA*" y "*CLASEA*" harán referencia a la misma etiqueta.
  Dado la posible existencia de dos identificadores con nombres similares (intercambiando mayúsculas con minúsculas), para solucionar este problema el módulo `GenCod` brindará un servicio `generarEtiqueta`, el cual devolverá un identificador único para poder aplicar en cada etiqueta a utilizar. De esta manera,

se evita que existan etiquetas duplicadas, y al mismo tiempo se distingue a qué método o clase se hace referencia en la generación de código.

- **Primario como lado izquierdo**:
  Como ya se mencionó anteriormente, cuando se realiza recorrido y chequeo de los nodos del AST, al mismo tiempo se realiza la generación de código.
  Una situación particular pasa con los nodos Primario, los cuales nos interesa saber si se ubican a la izquierda de una asignación. Como por ejemplo, supongamos que tenemos dos variables locales a y b, y que tenemos la siguiente asignación: `a = b;`.
  Notar que tanto `a` como `b` son `NodoVariable`, pero que la generación de código no será la misma para ambas. Para este caso, nos interesa hacer un `LOAD` con el *offset* de `b` (cargando su valor), mientras que para el caso de `a` haremos un `STORE` con su *offset* (guardando el valor del tope de la pila). Para poder evitar esta ambigüedad es necesario marcarle a cualquier nodo primario cuando están en un lado izquierdo de una asignación. Para ello, a `NodoExpPrimario` (y a los encadenados) se les agregó un atributo `esLadoIzquierdo` el cual originalmente iniciará en `false` para todos. Cuando se esté realizando el chequeo de sentencia de una asignación, al nodo que esté en el lado izquierdo se le marcará que tiene esa propiedad (estableciendo su atributo `esLadoIzquierdo` como `true`). Todos los nodos que tengan encadenados, le avisaran a su encadenado correspondiente su condición de lado izquierdo. Es decir, previo a la invocación del *encadenado.chequear()*, se le asigna un valor al atributo `esLadoIzquierdo` del encadenado, dependiendo del valor de `esLadoIzquierdo` del nodo que lo esté conteniendo.

- **Asignación de offsets**:
  Tanto variables como métodos necesitan de offset para saber su desplazamiento. Para los métodos, el *offset* marcará su desplazamiento dentro de la *Virtual Table* (VT) de su clase (o de clases heredadas). Los únicos métodos que necesitan un *offset* son los dinámicos, dado que para los estáticos se conoce su etiqueta en compilación. Sabiendo esto, los métodos estáticos permanecerán con *offset* -1, mientras que a los otros se les asignará su desplazamiento correspondiente. Por el lado de las variables, las de instancia necesitan saber su desplazamiento dentro del *Class Instance Record* (CIR), mientras que los parámetros y variables locales para el *Registro de Activación* (RA). Tanto para métodos (dinámicos) como para variables de instancia, el momento del cálculo de los *offset* es cuando se realiza la actualización de la tabla de métodos, dado que es necesario conocer la cantidad de atributos y de métodos que posee la clase ancestra para poder saber a partir de qué offset se le asigna a los nuevos atributos o métodos.
  En cuanto a los parámetros, la asignación del *offset* es más flexible, ya que la acción se puede realizar tanto en la creación de nodo, como en el chequeo de declaraciones o de sentencias. Lo que sí es necesario es tener los offset asignados previos a la generación del bloque del método (o constructor). En la implementación, la asignación de offsets para los parámetros se puede encontrar en el chequeo de sentencias, aunque como se mencionó, se puede trasladar dicha acción al chequeo de declaraciones.

Para las variables locales, el offset se va calculando y asignando a medida que se van declarando dentro del bloque. Es necesario notar que el offset de las variables locales será negativo, dado que se encuentran del otro lado del FP.

- **Casteo e InstanceOf**:
  Para los problemas de los operadores de casting e `instanceof`, es necesario remarcar que los *Class Instance Records* (CIR) de los objetos, además de reservar lugar para las variables de instancia, guardan 2 lugares más para tener una referencia a la *Virtual Table* (VT) y otro lugar para saber la clase asociada (ID de Clase según un identificador único de la clase). La operación para el instanceof es automática, dado que en MINIJAVA este operador es más restrictivo que en Java. Se compara el ID de la Clase del objeto (Usando `LOADREF 1`, ya que está en el segundo lugar del CIR), con el ID de la Clase a analizar (conocido en compilación). La comparación de dichos IDs devolverá un booleano que determinará si el objeto es instancia de la clase o no.
  Por el lado del casteo, la comparación en la misma, a diferencia de que necesito guardar la referencia en el tope de la pila para no perderla con el `LOADREF` (esto no pasaba en el `instanceof` ya que no nos interesaba).
  Vale la aclaración, que el casteo puede producir un error en ejecución. Para dicho error en ejecución se maneja mediante una etiqueta estática (y fija), que de alguna forma simula como si hubiera una excepción.

6.4. Decisiones sobre el diseño

- Por simplicidad, el método `main` que representa al programa principal, será elegido automáticamente por el compilador y mostrado en pantalla.
- Si no se especifica archivo de salida, el programa intentará crear un archivo con nombre idéntico al archivo de entrada, agregándole la extensión `.ceiasm`.
- No se realizan pruebas sobre instanciación sobre `null` ni tampoco de división por cero, ya que de dicha tarea se encargará la CEIVM.

6.5. Pruebas utilizadas

A diferencia de las etapas anteriores, para ésta etapa las pruebas fueron solamente de clase A. Dada la complejidad de la verificación en esta etapa, la batería de pruebas contendrá algunas específicas (las cuales requieren hacer una evaluación "manual"del código intermedio), mientras que otras serán del estilo de auto-verificables (no se revisará el código generado, sino que se examinará que como resultado de la ejecución del programa muestre por pantalla lo esperado).
En varios tests se utilizan "etiquetas" como [OK] y [ERROR], siendo [OK] un mensaje que se espera que aparezca, mientras que [ERROR] no debería figurar en ninguna ejecución.
Las pruebas comienzan desde lo simple, clases básicas con varios `main` para mostrar que se da advertencia por pantalla de la situación, así como también pruebas de la clase `System` para verificar que se están mostrando los valores correctamente por

pantalla. Luego de esto, se probó con algunas sentencias `if` y `while`, en dicho caso no solo se estaba probando la generación de código de esas estructuras, sino también las expresiones. Además, se hicieron pruebas para declaración de variables y para asignaciones (con y sin expresiones).
Luego, las pruebas fueron tomando complejidad, con `if` y `while` anidados, llamadas a métodos estáticos, pruebas sobre el construir una instancia de clase (usando `new`), llamadas a métodos dinámicos, pruebas sobre variables de instancia heredadas (para probar el correcto funcionamiento de los *offset*), expresiones de casteo e `instanceof`, entre otras cosas.
Lo bueno de las pruebas en esta étapa fue probar el correcto funcionamiento de programas lo suficientemente complejos para notar la fortaleza del compilador.