Proyecto Compiladores e Interpretes

Etapa 5

Generación de código para la CeIVM

Rodrigo Díaz Figueroa

LU 97400

Clases utilizadas y decisiones de diseño

Cambios desde la última etapa:

Se agregaron las clases correspondientes al AST y Operadores y Tipos.

Nuevas clases:

Correspondientes al AST:

***Sentencias*:**

NodoSentencia
NodoBloque
NodoDeclaracionVars
NodoIf
NodoWhile
NodoSentenciaSimple
NodoPuntoComa
NodoAsignacion
NodoReturn

Expresiones:

NodoExpresion
NodoExpBin
NodoExpUn
NodoOperando
NodoPrimario
NodoLiteral
NodoLitEnt
NodoLitNull
NodoLitBool
NodoLitChar
NodoLitString
NodoLlamDirect
NodoLlamEstat
NodoExpParent
NodoConst
NodoConstArray
NodoConstComun
NodoAcceso
NodoVar
NodoThis

Encadenados:

Encadenado
NodoVarEncad
NodoLlamEncad
NodoArregloEncad

Operadores:

Operador
OpMat
OpBool
OpComp
OpCompMat

Tipos:

Tipo
TipoVoid
TipoReferencia
TipoInt
TipoBool
TipoChar
TipoString
TipoClase
TipoNull

Decisiones de diseño:

- Para las variables locales a los bloques, se reservan espacio antes de generar el código del bloque y se liberan 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 ningún método dinámico 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.

Logros

Los logros que se intentan alcanzar de esta etapa son:

- **Imbatibilidad Generación.**
- **Generando arreglos de todo.**
- **Inicializaciones inline generadas.**
- **Generación sobrecargadas.**

Compilación y Ejecución

Para compilar el programa desde consola se deberá ejecutar el siguiente comando

```
>javac Minijavac.java
```

Observación: Si este comando no compila todas las clases del programa se deberá ejecutar entonces

```
>javac Minijavac.java AnalizadorSintactico.java AnalizadorLexico.java
Utl.java Token.java EntradaSalida.java LexicoException.java
SintacticException.java Tipo.java . . . (todas las clases)
```

Para la ejecución del programa desde consola se deberá ejecutar

```
>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.

El programa soporta paths relativos del archivo fuente si se encuentra en el mismo directorio que el resto de los archivos del programa.

El archivo de salida sera creado (o reescrito) en el mismo directorio que se encuentra el programa.

El archivo de salida por defecto sera “out.txt” en el caso de que no se especifique uno, y se creara en el mismo directorio donde se encuentra el programa.

Ejemplos:

```
>java Minijavac ./Test/Correctos/Test1.java
```

```
>java Minijavac ./Test/Correctos/Test1.java out.ceiasm
```

El archivo Test1.java debe estar dentro del directorio Test/Correctos/ que se encuentra en el mismo directorio que los archivos fuente

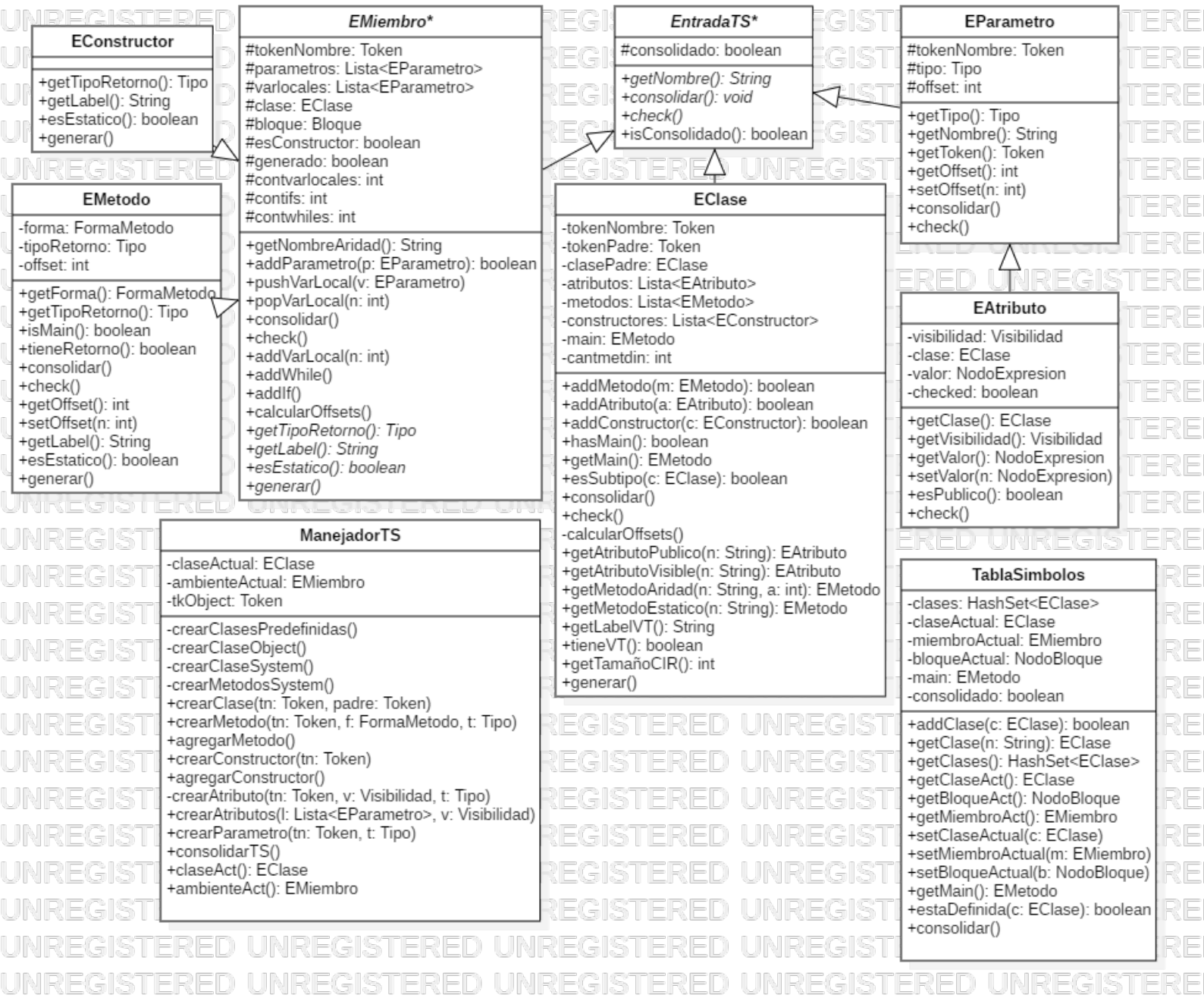
```
>java Minijavac C:/Testing/Incorrectos/Test2.java
```

```
>java Minijavac C:/Testing/Incorrectos/Test2.java Test2.out
```

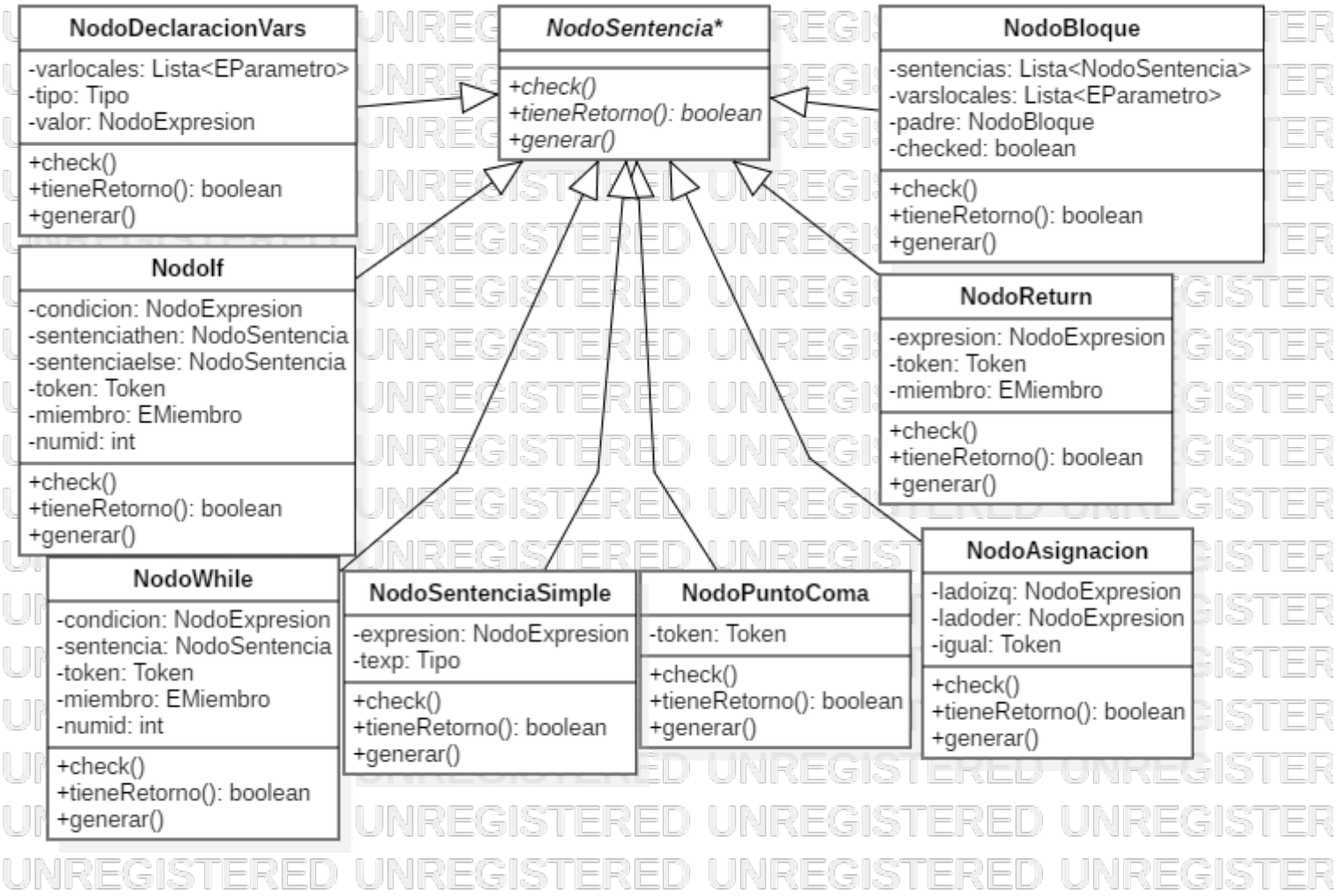
El archivo Test2.java debe estar dentro del directorio que especifica el path absoluto

Diagramas de clases.

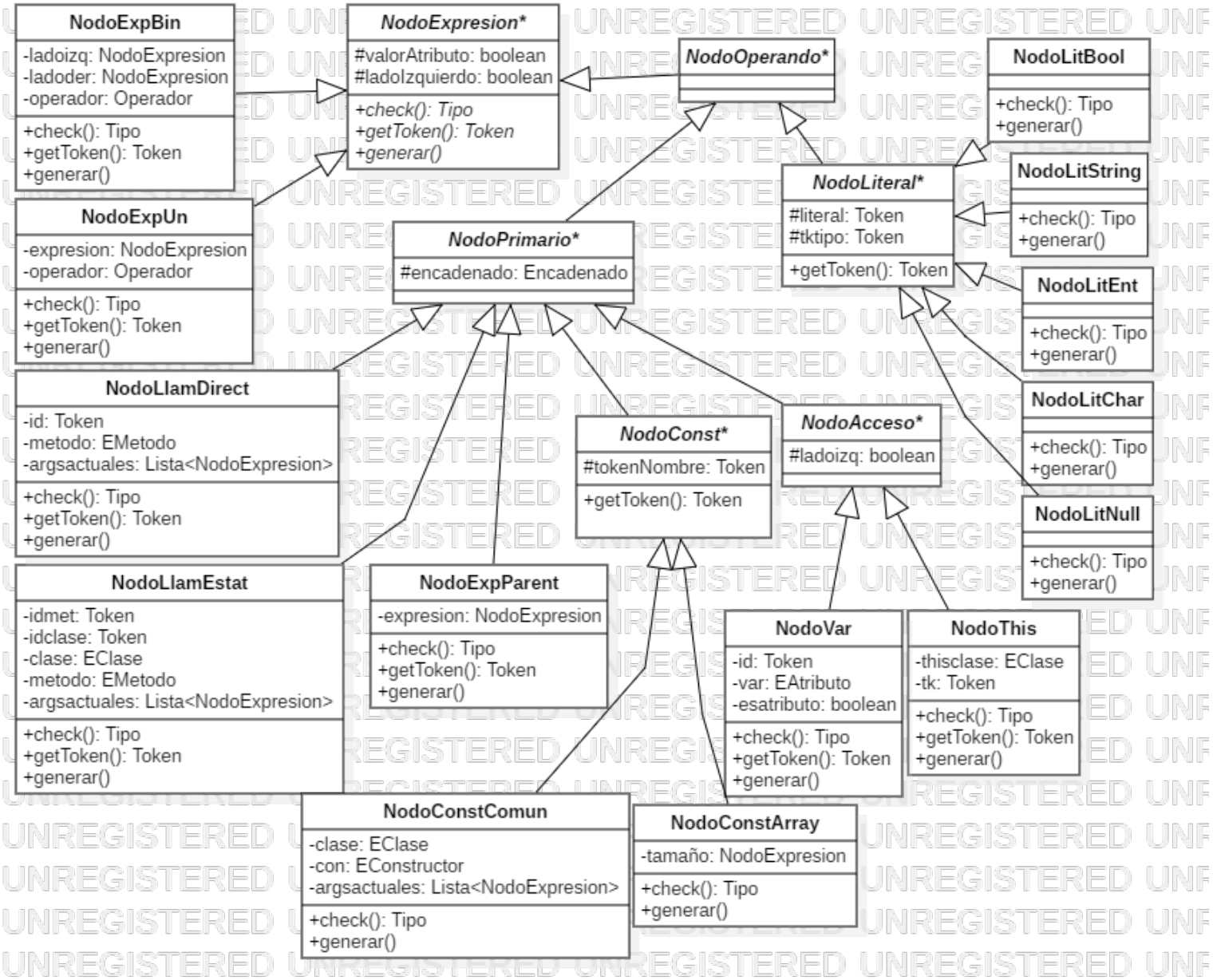
Tabla de símbolos:



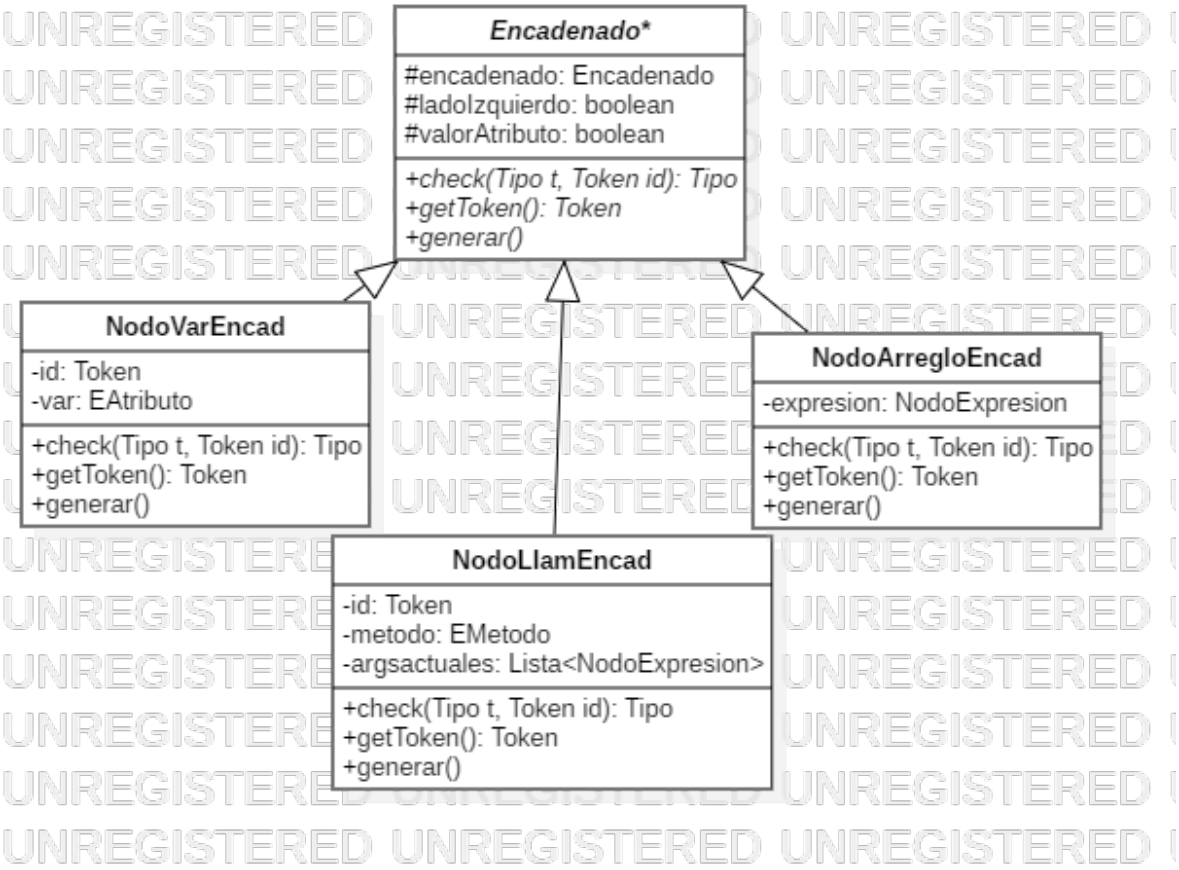
Sentencias:


NodoDeclaracionVars
-varlocales: Lista<EParametro>
-tipo: Tipo
-valor: NodoExpresion
+check()
+tieneRetorno(): boolean
+generar()
NodoSentencia*
+check()
+tieneRetorno(): boolean
+generar()
NodoBloque
-sentencias: Lista<NodoSentencia>
-varslocales: Lista<EParametro>
-padre: NodoBloque
-checked: boolean
+check()
+tieneRetorno(): boolean
+generar()
NodoIf
-condicion: NodoExpresion
-sentenciathen: NodoSentencia
-sentenciaelse: NodoSentencia
-token: Token
-miembro: EMiembro
-numid: int
+check()
+tieneRetorno(): boolean
+generar()
NodoReturn
-expresion: NodoExpresion
-token: Token
-miembro: EMiembro
+check()
+tieneRetorno(): boolean
+generar()
NodoWhile
-condicion: NodoExpresion
-sentencia: NodoSentencia
-token: Token
-miembro: EMiembro
-numid: int
+check()
+tieneRetorno(): boolean
+generar()
NodoSentenciaSimple
-expresion: NodoExpresion
-texp: Tipo
+check()
+tieneRetorno(): boolean
+generar()
NodoPuntoComa
-token: Token
+check()
+tieneRetorno(): boolean
+generar()
NodoAsignacion
-ladoizq: NodoExpresion
-ladoder: NodoExpresion
-igual: Token
+check()
+tieneRetorno(): boolean
+generar()

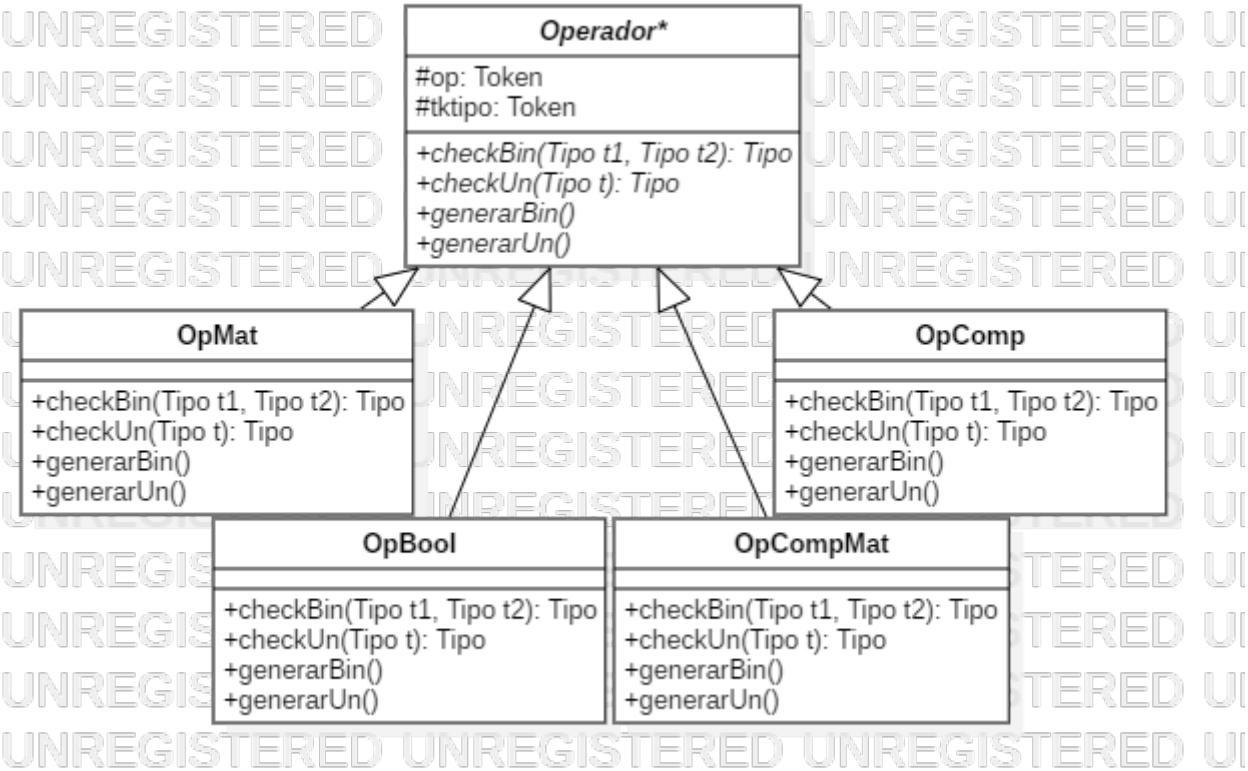
Expresiones:



Encadenados:



Operadores:



Calculo de offsets y etiquetas:

El calculo de offsets de los métodos dinámicos, atributos de clase y los parametros 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 metodo para obtenerlas.

Las etiquetas estan 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 miembo y <etiquetamiebro> 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 miembo y <etiquetamiebro> 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 metodo para obtener su etiqueta si es estatico u obtener su offset si es dinamico

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 dinamico

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

Testing

Para el testing del programa se utilizaron varios casos de prueba correctos. Con estos se intenta cubrir todos los casos que pueden llegar a ser mas problemáticos para la generación de código.
Cada caso de test contiene al principio un comentario con la explicación para la cual esta hecho dicho caso.