COMPILADORES E INTÉRPRETES

SEGUNDO CUATRIMESTRE DE 2016

MANUAL TÉCNICO

COMPILADOR MINIJAVA

Matias Marzullo, 80902

Índice

- 4. Introducción
- 5. **1. Analizador Léxico**
  - 5. Compilar y Ejecutar
  - 5. Alfabeto
  - 6. Tokens
  - 7. Errores Léxicos Detectados
    - 7. Errores Generales
    - 7. Errores Detectados
  - 8. Catálogo de casos de Test
    - 8. Test Válidos
    - 8. Test Inválidos
  - 9. Diseño
    - 9. Decisiones Generales
  - 9. Especificación de Clases
    - 9. AnalizadorLexico.java
    - 9. Token.java
    - 9. Principal.java
- 10. **2. Analizador Sintáctico**
  - 11. Proceso de transformación
    - 11. Eliminación de los elementos de la gramática EBNF
    - 13. Eliminación de recursión a izquierda
    - 15. Factorización
  - 17. Ambigüedad de la gramática
  - 19. Sentencias con un método de análisis sintáctico LR
  - 19. Expresiones de tipo Casting
  - 20. Errores Sintácticos Detectados
  - 20. Diseño: Analizador Sintáctico
- 21. **3. Analizador Semántico**
  - 21. Esquema de Traducción (EDT)
  - 27. Diagramas de Clases

- 31. Errores Semánticos Detectados
- 32. Explicación de los Métodos de la Tabla de Simbolos
- 33. AST (Abstract Syntax Tree)
- 33. Chequeos en Clase.java
- 33. Chequeos en Metodo.java
- 34. Diseño: Analizador Semántico

❖ 35. **4. Generación de Código CeIVM**

- 35. Control de Declaraciones
- 35. Control de Sentencias
- 36. Generación de Código

Introducción

En este informe se presentará la implementación completa de un compilador para el lenguaje MINIJAVA.

Dicho compilador fue realizado en diferentes etapas en las que se distinguen el desarrollo del Analizador Léxico, el Analizador Sintáctico, el Analizador Semántico y la Generación de Código.

Cada una de estas etapas será desarrollada en su totalidad en el presente informe.

1. Analizador Léxico

La implementación del analizador léxico se llevó a cabo haciendo uso del lenguaje de programación JAVA. Dicho analizador convierte una entrada de caracteres en una entrada de Tokens, donde cada Token se corresponde con un lexema de MINIJAVA, además, cada Token tiene asociado un número de línea de código en la cual el mismo fue encontrado.

Compilar y Ejecutar

- Para compilar el código por consola, sin generar un *.jar,* se realiza:

```
$ javac alex/Principal.java
```

- Una vez compilado la sintaxis para ejecutar el analizador Léxico es:

<PROGRAM_NAME> <INPUT_FILE> [<OUTPUT_FILE>]

Donde <PROGRAM_NAME> indica el nombre del ejecutable, el nombre asignado a la clase que contiene el *main* se llama **Principal**.
<INPUT_FILE> es el archivo a analizar léxicamente (el usuario deberá poder elegir qué archivo analizar
<OUTPUT_FILE> es un parámetro opcional que de estar presente especifica el archivo de salida donde se volcará el listado de tokens, lexemas y número de línea generado. Si se omite este parámetro, el listado será presentado en pantalla.
Ejemplos:

`$ java alex.Principal prueba.java output.txt` *(JAVA, salida en el archivo output.txt)*

`$ java alex.Principal prueba2.java` *(JAVA, salida por pantalla)*

Alfabeto

El alfabeto utilizado para MINIJAVA es el código ASCII:

*! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?*
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

Tokens

TOKEN	EXPRESION REGULAR
intLiteral	*(0...9)**
IdClase	*(A..Z)(A..Z + a...z + 0..9 + _)**
idMetVar	*(a..z)(A..Z + a...z + 0..9 + _)**
(|) | { | } | ; | , | . | [|]	*(|) | { | } | ; | , | . | [|]*
> | < | ! | == | >= | <= | !=	*> | < | ! | == | >= | <= | !=*
= | + | - | * | / | && | || | %	= | + | - | * | / | && | || | %
EOF	*EOF*
extends	*extends*
class	*class*
void	*void*
boolean	*boolean*
if	*if*
else	*else*
this	*this*
new	*new*
static	*static*
dynamic	*dynamic*
public	*public*
charLiteral	*char*
int	*int*
stringLiteral	*String*
while	*while*
return	*return*
instanceof	*instanceof*
null	*null*
true	*true*
false	*false*
private	*private*

Errores Léxicos Detectados

Errores generales

La clase Principal define un conjunto de errores generales ajenos al analizador. Estos errores son:

“`Se esperaban argumentos`”: Surge cuando no se introduce un archivo a analizar: <INPUT_FILE>
“`Demasiados argumentos`”: Aparecerá cuando se introducen 3 o más archivos.
“`El archivo de entrada es inválido`”: Si surge algún problema con los archivos de entrada.

Errores Detectados:

El analizador es capaz de reconocr los errores léxicos que se enumeran a continuación:

- **CharacterLiteral**: Caracteres mal formados, por ejemplo cuando se intenta incluir más de un caracter entre las comillas simples (quotes) que delimitan un caracter. Son excepción a este error los caracteres escapados, tabs (\t) y saltos de línea (\n):

```
"El literal caracter '"+readed+"' es invalido."
```

- **Character**: Caracteres que no pertenecen al alfabeto:

```
"El caracter '"+readed+"' es invalido."
```

- **Comment**: Comentarios multilinea sin cerrar:

```
"Error de comentario."
```

- **NumberLiteral**: El valor de un literal entero no corresponde a su interpretación estándar en base 10:

```
"El literal numero '"+readed+"' es invalido."
```

- **StringLiteral:** Cadenas (String) mal formadas, por ejemplo, cuando se incluye un salto de línea antes del cierre de la cadena:

```
"El literal String \'"+readed+"\' es invalido."
```

La ejecución del analizador termina inmediatamente cuando un error es encontrado.

```
Error lexico: compilacion terminada.
```

Catálogo de casos de Test

Test Válidos

- TEST1Componentes.txt: Se presentan todos los componentes, expresiones regulares del compilador léxico MINIJAVA.
- TEST2identificadores.txt: se listan las combinaciones posibles de identificadores, tanto de clase como de métodos y variables.
- TEST3enteros: se muestran expresiones de enteros válidas.
- TEST4LitCar.txt se muestran los literales caracteres.

Test Inválidos

Los errores Léxicos detectados, mostrados en la sección anterior se comprueban con los siguientes Test:

- TEST5litCarac.txt: Caracteres Literales no válidos. Error: **CharacterLiteral**
- TEST6caracter.txt: Muestra un caracter no válido. Error: **Character**
- TEST7comentarios: Muestra como falla el compilador con los comentarios. Error: **Comment**
- TEST8numeros.txt: Muestra cuando un número está mal formado. **NumberLiteral**
- TEST9string: muestra cuando una cadena está mal formada. Error: **StringLiteral**

Diseño: Analizador Léxico

Decisiones Generales

Como primera medida, se decidió optar, para la implementación general del analizador, por el enfoque en la eficiencia del mismo en vez de elegir un diseño más modularizado. Por esto, el método que se utiliza para obtener cada uno de los tokens de archivo fuente cuenta con todas las decisiones dentro de la misma, sin tener que invocar a una nueva unidad por cada caso.

Para la lectura del archivo, se decidió utilizar la clase provista por JAVA (*BufferedReader*) el cual lee de a un caracter.

El programa fue programado en el sistema operativo mint.

Especificación de Clases

AnalizadorLexico.java

Esta clase es la encargada de llevar a cabo el análisis léxico del archivo fuente, cuenta con un método *inicializarPR*, el cuál procederá a cargar el mapeo necesario para identificar todas las posibles palabras reservadas. Y un método *getToken*, que retorna un objeto de tipo Token (especificado más adelante) cuyo comportamiento es el siguiente: Dentro de un ciclo *while*, el cual continuará mientras no se haya completado un token o se encuentre un error.

También cuenta con un método para obtener el próximo carácter: *nextChar* y otro encargado de procesar los errores: *error.*

Token.java

Esta clase se utiliza para representar los Token que se encuentran dentro del archivo fuente, el cual se identifica por código (representa el tag que identifica a los Token), el lexema asociado al Token y la línea en la que fue encontrado.

Cuenta además, con métodos para retorna el nombre que identifica al Token, y un método que retorna el lexema formado asociado al Token y un método que retorna el número de línea en la cual el Token se encuentra.

Principal.java

Esta clase es la clase “main” del proyecto, es la encargada de realizar los controles acerca de la cantidad de argumentos ingresados y la validez de los mismos (como puede ser una ruta de archivo incorrecta), la alineación en forma de columnas de los Tokens a mostrar y la creación del archivo de salida, si esto fuera requerido en la invocación.

2. Analizador Sintáctico

Para la implementación del Analizador Sintáctico primero se llevó a cabo la eliminación de la recursión a izquierda y posterior factorización de la gramática presentada por la cátedra para de esta manera poder desarrollar el analizador sintáctico de forma de que el mismo sea descendente, predictivo y recursivo.

A partir de la gramática,

- Se llevara a cabo un análisis para mostrar por qué esta no cumple con la propiedad de ser LL(1).
- Se mostrarán tres producciones de la gramática original de MINIJAVA que hubiesen sido más fáciles de tratar con un método de análisis sintáctico LR.
- Y se tratará el por qué las expresiones de tipo casting en MINIJAVA fueron modeladas utilizando “[“ y “]” en vez de “(“ y “)” como lo hace JAVA.

Luego, se describirán cada uno de los errores que el analizador es capaz de detectar, especificando las causas de las mismas.

Por último, se detallaran todas las decisiones de diseño que se tuvieron en cuenta para la implementación del analizador sintáctico. Además, se hará una descripción general de cada clase, que incluye la implementación y que no hayan sido descriptas en el analizador léxico o aquellas que hayan sido modificadas.

Proceso de transformación

Eliminación de los elementos de la gramática EBNF

<Inicial> → <Clase> <Inicial> | <Clase>
<Clase> → class idClase <Herencia> { <MiembroL>} | class idClase {<miembroL>}
<MiembroL> → <Miembro> <MiembroL> | λ
<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> | <ArgFormal> , <ListaArgsFormales>
<ArgFormal> → <Tipo> idMetVar
<FormaMetodo> → static | dynamic
<TipoMetodo> → <Tipo> | void
<Tipo> → <TipoPrimitivo> | idClase
<TipoPrimitivo> → boolean | char | int | String
<ListaDecVars> → idMetVar | idMetVar , <ListaDecVars>
<Bloque> → {<SentenciaL>}
<SentenciaL> → <Sentencia> <SentenciaL> | λ
<Sentencia> → ;
<Sentencia> → <Asignación> ;
<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> | <ExpAd> instanceof idClase | <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>

$<Operando> \rightarrow <Primario>$
$<Literal> \rightarrow null \mid true \mid false \mid intLiteral \mid charLiteral \mid StringLiteral$

$<Primario> \rightarrow (<Expresion>)\ <LlamadaoIdEnc>$
$<Primario> \rightarrow this\ <LlamadaoIdEnc>$
$<Primario> \rightarrow idMetVar\ <LlamadaoIdEnc>$
$<Primario> \rightarrow (idClase\ .\ <Llamada>\ )\ <LlamadaoIdEnc>$
$<Primario> \rightarrow new\ idClase\ <ArgsActuales>\ <LlamadaoIdEnc>$
$<Primario> \rightarrow <Llamada>\ <LlamadaoIdEnc>$

$<LlamadaoIdEnc> \rightarrow <LlamadaoIdEncadenado>\ <LlamadaoIdEnc> \mid \lambda$

$<LlamadaoIdEncadenado> \rightarrow\ .\ <Llamada> \mid .\ idMetVar$
$<Llamada> \rightarrow idMetVar\ <ArgsActuales>$

$<ArgsActuales> \rightarrow (\ <ListaExps>\ ) \mid (\ )$
$<ListaExps> \rightarrow <Expresion> \mid <Expresion>\ ,\ <ListaExps>$

Eliminación de recursión a izquierda

<Inicial> → <Clase> <Inicial> | <Clase>
<Clase> → class idClase <Herencia> { <ListaMiembros>} | class idClase {<ListaMiembros>}
<ListaMiembros> → <Miembro> <ListaMiembros> | λ
<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> | <ArgFormal> , <ListaArgsFormales>
<ArgFormal> → <Tipo> idMetVar
<FormaMetodo> → static | dynamic
<TipoMetodo> → <Tipo> | void
<Tipo> → <TipoPrimitivo> | idClase
<TipoPrimitivo> → boolean | char | int | String
<ListaDecVars> → idMetVar | idMetVar , <ListaDecVars>
<Bloque> → { <SentenciaL>}
<SentenciaL> → <Sentencia> <SentenciaL> | λ
<Sentencia> → ;
<Sentencia> → <Asignación> ;
<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> <ExprP>
<ExprP> → || <ExpAnd> <ExprP> | λ
<ExpAnd> → <ExpIg> <ExpAndP>
<ExpAndP> → && <ExpIg> <ExpAndP> | λ
<ExpIg> → <OpComp> <ExpIgP>
<ExpIgP> → <OpIg> <Expcomp> <ExpIgP> | λ
<ExpComp> → <ExpAd> <OpComp> <ExpAd> | <ExpAd> instanceOf idClase | <ExpAd>

<ExpAd> → <ExpMul> <ExpAdP>
<ExpAdP> → <OpAd> <ExpMul> <ExpAdP> | λ
<ExpMul> → <ExprUn> <ExpMulP>
<ExpMulP> → <OpMul> <ExprUn> <ExpMulP> | λ
<ExpCast> → [idClase] <Operando> | <Operando>

<OpIg> → == | !=
<OpComp> → < | > | <= | >=
<OpAd> → + | -
<OpUn> → + | - | !

*<OpMul> → * | / | %*
<Operando> → <Literal>
<Operando> → <Primario>
<Literal> → null | true | false | intLiteral | charLiteral | stringLiteral

<Primario> → (<Expresion>) <LlamadaoIdEnc>
<Primario> → this <LlamadaoIdEnc>
<Primario> → idMetVar <LlamadaoIdEnc>
<Primario> → (idClase . <Llamada>) <LlamadaoIdEnc>
<Primario> → new idClase <ArgsActuales> <LlamadaoIdEnc>
<Primario> → <Llamada> <LlamadaoIdEnc>

<LlamadaoIdEnc> → <LlamadaoIdEncadenado> <LlamadaoIdEnc> | λ
<LlamadaoIdEncadenado> → . <Llamada> | . idMetVar
<Llamada> → idMetVar <ArgsActuales>

<ArgsActuales> → (<ListaExps>) | ()
<ListaExps> → <Expresion> | <Expresion> , <ListaExps>

Factorización

<Inicial> → <Clase> <ListaClases>
<ListaClases> → <Inicial> | λ

<Clase> → class idClase <Herencia> { <ListaMiembros> }
<Herencia> → extends idClase | λ

<ListaMiembros> → <Miembro> <ListaMiembros> | λ
<Miembro> → <Atributo> | <Ctor> | <Metodo>
<Atributo> → <Visibilidad> <Tipo> <ListaDecVars> ;
<Metodo> → <FormaMetodo> <TipoMetodo> idMetVar <ArgsFormales> <Bloque>
<Ctor> → idClase <ArgsFormales> <Bloque>
<Visibilidad> → public | private

<ArgsFormales> → (<ArgsFormalesP>
<ArgsFormalesP> → <ListaArgsFormales>) |)
<ListaArgsFormales> → <ArgFormal> <ListaArgsFormalesP>
<ListaArgsFormalesP> → , <ListaArgsFormales> | λ

<ArgFormal> → <Tipo> idMetVar
<FormaMetodo> → static | dynamic
<TipoMetodo> → <Tipo> | void
<Tipo> → <TipoPrimitivo> | idClase
<TipoPrimitivo> → boolean | char | int | String
<ListaDecVars> → idMetVar <ListaDecVarsP>
<ListaDecVarsP> → , <ListaDecVars> | λ
<Bloque> → {<ListaSentencias>}
<ListaSentencias> → <Sentencia> <ListaSentencias> | λ

<Sentencia> → ;
<Sentencia> → <Primario> <Asignacion> ;
<Sentencia> → <Tipo> <ListaDecVars> ;
<Sentencia> → if (<Expresion>) <Sentencia> <SentenciaElse>
<Sentencia> → while (<Expresion>) <Sentencia>
<Sentencia> → <Bloque>
<Sentencia> → return <SentenciaP> ;
//Fue factorizado <Cuerpo>
<SentenciaElse> → else <Sentencia> | λ
<SentenciaP> → <Expresion> | λ

// fue factorizado <SentenciaLlamada>
<Asignacion> → = <Expresion> | λ
<Expresion> → <ExpAnd> <ExprP>
<ExprP> → || <ExpAnd> <ExprP> | λ
<ExpAnd> → <ExpIg> <ExpAndP>
<ExpAndP> → && <ExpIg> <ExpAndP> | λ
<ExpIg> → <OpComp> <ExpIgP>
<ExpIgP> → <OpIg> <Expcomp> <ExpIgP> | λ

<ExpComp> → <ExpAd> <ExpCompP>
<ExpCompP> → <OpComp> <ExpAd> | instanceOf idClase | λ

<ExpAd> → <ExpMul> <ExpAdP>
<ExpAdP> → <OpAd> <ExpMul> <ExpAdP> | λ
<ExpMul> → <ExprUn> <ExpMulP>
<ExpMulP> → <OpMul> <ExprUn> <ExpMulP> | λ
<ExpCast> → [idClase] <Operando> | <Operando>

<OpIg> → == | !=
<OpComp> → < | > | <= | >=
<OpAd> → + | -
<OpUn> → + | - | !
*<OpMul> → * | / | %*
<Operando> → <Literal> | <Primario>
<Literal> → null | true | false | intLiteral | charLiteral | stringLiteral

<Primario> → (<PrimarioP>) <LlamadaoIdEnc>
<Primario> → this <LlamadaoIdEnc>
<Primario> → idMetVar <PrimarioPP> <LlamadaoIdEnc>
<Primario> → new idClase <ArgsActuales> <LlamadaoIdEnc>
<PrimarioP> → <Expresion> | idClase . <Llamada>
<PrimarioPP> → <ArgsActuales> | λ

<LlamadaoIdEnc> → <LlamadaoIdEncadenado> <LlamadaoIdEnc> | λ

<LlamadaoIdEncadenado> → . idMetVar <LlamadaoIdEncadenadoP>
<LlamadaoIdEncadenadoP> → <ArgsActuales> | λ
<Llamada> → idMetVar <ArgsActuales>

<ArgsActuales> → (<ArgsActualesP>
<ArgsActualesP> → <ListaExps>) |)

<ListaExps> → <Expresion> <ListaExpsP>
<ListaExpsP> → , <ListaExps> | λ

Ambigüedad de la gramática

La gramática obtenida como resultado de las transformaciones de eliminación de recursividad a izquierda y factorización no es de tipo LL(1) ya que es ambigua. La ambigüedad se da en las siguientes reglas de producción:

$$<Sentencia> \rightarrow if\ <Cuerpo>\ <SentenciaElse>$$

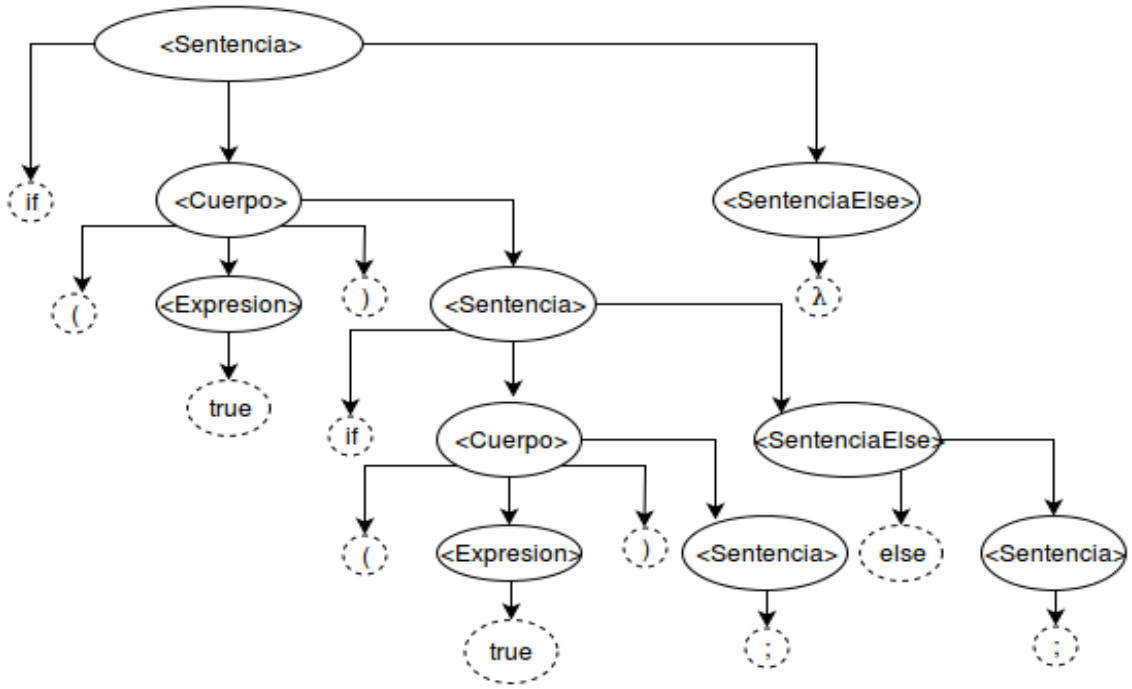
$$<Cuerpo> \rightarrow (<Expresion>)\ <Sentencia>$$

$$<SentenciaElse> \rightarrow else\ <Sentencia> \mid \lambda$$

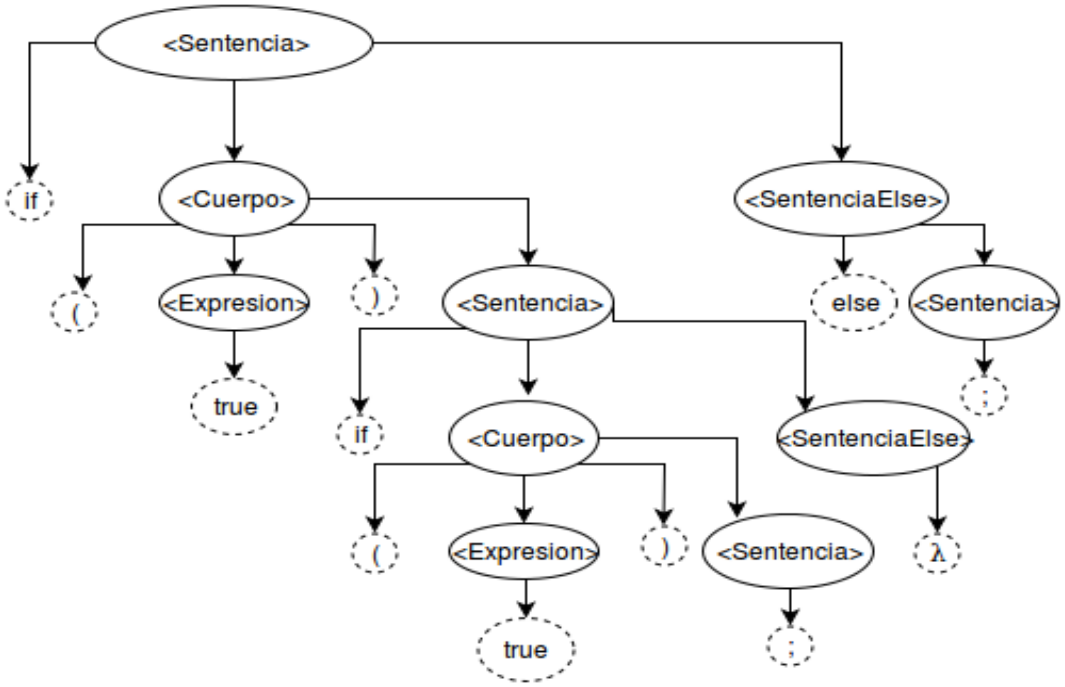
Justificación

Para la cadena *"if (true) if (true); else ;"* que representa una *<sentencia>*, es posible obtener dos árboles de derivación distintos. El problema es que no se especifica en la gramática a que "*if*" corresponde el primer "*else*" encontrado.

1) Primer árbol de derivación:



2) Segundo árbol de derivación:



Solución en la implementación

En la implementación se soluciona la ambigüedad de forma tal que siempre se corresponde con el árbol de derivación (1). Esto es que el primer "**else**" siempre se corresponde con el "**if**" más cercano.

Sentencias con un método de análisis sintáctico LR

Los analizadores sintácticos LR son controlados por tablas, en forma muy parecida a los analizadores sintácticos LL no recursivos.

El análisis sintáctico LR es atractivo por una variedad de razones:

- Pueden reconocer la inmensa mayoría de los lenguajes de programación que puedan ser generados mediante gramáticas libres de contexto.
- **Ventaja**: localiza un error sintáctico casi en el mismo instante en que se produce con lo que se adquiere una gran eficiencia de tiempo de compilación frente a procedimientos menos adecuados como puedan ser los de retroceso. Además, los mensajes de error indican con precisión la fuente del error.

Desventaja: es demasiado trabajo construir un analizador sintáctico LR en forma manual para una gramática común de un lenguaje de programación. Se necesita una herramienta especializada: un generador de analizadores sintácticos LR.

Elegiría gramáticas que no tengan conflicto desplazamiento-reducción. Y que pasan por muchos métodos hasta encontrar el resultado, como son:

$$<ExpOr> \rightarrow <ExpOr> \ // <ExpAnd> \mid <ExpAnd>$$

$$<ExpAnd> \rightarrow <ExpAnd> \ \&\& <ExpIg> \mid <ExpIg>$$

$$<ExpIg> \rightarrow <ExpIg> <OpIg> <ExpComp> \mid <ExpComp>$$

Expresiones de tipo *Casting*

Lo primero que se observa es que entraría en conflicto ExpCast:

$$\{Opción\ 1\} \qquad \{Opción\ 2\}$$

$$<ExpCast> \rightarrow (\ idClase\ ) <Operando> \mid <Operando>$$

$$<Operando> \rightarrow <Primario>$$

$$<Primario> \rightarrow (\ <Expresion>\ ) <LlamadaoIdEncadenado>^{*}$$

$$<Primario> \rightarrow (\ idClase\ .\ <Llamada>\ ) <LlamadaoIdEncadenado>^{*}$$

Ya que siempre entraría por la “opción 1” y nunca por la “opción 2”. Es decir, no hay una forma de identificador que expresión estoy invocando ya que las dos (ExpCast y Primario) comienzan con el terminal “(“.

Lo otro es que, aunque se puede resolver el problema anterior en el proceso de transformación, habría que factorizar demasiado y esto ocasionaría roturas en la gramática.

Errores Sintácticos Detectados

El analizador sintáctico desarrollado es capaz de detectar todos los errores sintácticos. Al producirse dicho error, se arroja una excepción de tipo “SintaxError”, y el mismo indica al usuario en que línea se produjo el error, que se esperaba y que se encontró.

Diseño: Analizador Sintáctico

Decisiones Generales

Para la implementación general, se creó una clase (AnalizadorSyn) que contiende un método por cada no-terminal de la gramática de MINIJAVA, por lo que queda una clase bien modularizada y permite seguir el flujo a través de ella.
El análisis comienza en el método “analize()” cuya funcionalidad es invocar al método “inicial()” que es el encargado de realizar el análisis de los encabezados y las clases, respectivamente; hasta alcanzar el Token EOF correspondiente al final del archivo.

Para llevar el control de los Token que el analizador espera encontrar, se implementó un método denominado “match()”, que recibe como parámetro el identificador (string) de Token que se espera, Compara el token encontrado con lo que se esperaba de acuerdo a la gramática. En caso de encontrar un token no esperado o el fin de archivo durante el proceso de análisis, se procederá a detenerlo y a notificar el error; si coincidiera, se le pide al analizador léxico un nuevo Token, y en analizador sintáctico actualiza su Token.

Para esta etapa en la clase Token se le agregó un método para hacer las comparaciones (equals) más legibles en el compilador sintáctico.

La clase Principal se modificó y solo controla que esté el archivo de origen y que sea correcto.

3. Analizador Semántico

En primer lugar se presenta el Esquema de Traducción Semántico completo. En este caso las reglas semánticas se añaden delimitadas de la siguiente forma:

{Regla Semántica}

Se muestran los diagramas de clases utilizados para modelar la tabla de símbolos y las estructuras auxiliares utilizadas. Asi como también los errores detectados, se hará una explicación de los métodos de la TS, AST, chequeos en Clase.java y Metodo.java. Por último se mensionaran algunoas consideraciones sobre el diseño

Esquema de Traducción (EDT)

El siguiente esquema de traducción fue construido a partir de la gramática reducida a izquierda y factorizada. En él, se encuentran todas las acciones destinadas a la construcción del AST y la Tabla de Símbolos del compilador.

```
<Inicial> → <Clase> <ListaClases>
<ListaClases> → <Inicial> | λ
<Clase> → class idClase <Herencia>
        {
        if(!ts.estaClase(identificador.getLexema())){
                        ts.addClase(identificador);
                        clase.herencia = setHereda.nombre;
                }
                else
                        throw new ExceptionSem("Error Semantico: Se declaró nuevamente la clase");
        }
        { <ListaMiembros> }
<Herencia> → extends idClase
                {<Herencia>.nombre = <Herencia>.nombre}}|
                  λ {<Herencia>.nombre = "Object"}
<ListaMiembros> → <Miembro> <ListaMiembros> | λ
<Miembro> → <Atributo> | <Ctor> | <Metodo>
<Atributo> → <Visibilidad> <Tipo> <ListaDecVars> {
                                <Atributo>.v = <Visibilidad>.nombre
                                <Atributo>.tipo = <Tipo>.tip   }
            { <ListaDecVars>.lista = new lista () }
                <listaDecVars> {
                        Para cada var de <listaDecVars>.lista ts.getClaseActual().addVariable(var,
                                                        <Atributo>.v, <Atributo>.tipo)
            } ;
<Metodo> → <FormaMetodo> <TipoMetodo> idMetVar
                {
                Metodo m = new Metodo(<idMetVar>.lexema, <Metodo>.f, <Metodo>.tipo,
                                        <Metodo>.esFinal, <Metodo>.v);
                ts.getClaseActual().addMetodo(m);
```

```
                m.setClase(ts.getClaseActual());
                ts.setMetodoActual(m);
}
        <ArgsFormales> <Bloque> {m.setCuerpo(<Bloque>.salida)}

<Ctor> → idClase {
                if(!id.getLexema().equals(ts.getClaseActual().getNombre())){
                        throw new Exception("Error semantico: el constructor no tiene
                                        el mismo nombre que la clase.");    }
                if(!ts.getClaseActual().getTieneConst()){
                Metodo m =
                        ts.getClaseActual().addConstructor(aux,ts.getClaseActual()); ts.setMetodoActual(m);
                }
                else {
                        throw new Exception("Error semantico: la clase ya tiene un constructor definido);
                }
}
        <ArgsFormales>
        {ts.getMetodoActual().setCuerpo(<Bloque>.salida );}
<Visibilidad> → public { <Visibilidad>.vis ⇐ "public" } |
                  private { <Visibilidad>.vis ⇐ "private" }

<ArgsFormales> → ( <ArgsFormalesP>
<ArgsFormalesP> → <ListaArgsFormales> ) | )
<ListaArgsFormales> → <ArgFormal> <ListaArgsFormalesP>
<ListaArgsFormalesP> → , <ListaArgsFormales> | λ

<ArgFormal> → <Tipo> idMetVar
                {ts.getMetodoActual().addParametro(idMetVar.lexema, tipo.salida);}
<FormaMetodo> → static { <FormaMetodo>maMetodo.forma ⇐ "static" } |
                  dynamic { <FormaMetodo>.forma ⇐ "dynamic" }
<TipoMetodo> → <Tipo>  { <TipoMetodo>.tipo ⇐ <Tipo>.salida } |
                  void { <TipoMetodo>.tipo ⇐ new void() }
<Tipo> → <TipoPrimitivo> { <Tipo>.salida ⇐ <TipoPrimitivo>.salida } |
          idClase { <Tipo>.salida ⇐ new TipoClase (idClase.lexema) }
<TipoPrimitivo> → boolean { tipoPrimitivo.salida ⇐ new TipoBool() } |
                    char { <TipoPrimitivo>.salida ⇐ new TipoChar() } |
                    int  { <TipoPrimitivo>.salida ⇐ new TipoInt() } |
                    String { <TipoPrimitivo>.salida ⇐ new TipoString() }
<ListaDecVars> → idMetVar <ListaDecVarsP>
<ListaDecVarsP> → , <ListaDecVars> | λ
<Bloque> → {  {
                        <Bloque>.b ⇐ new Bloque();
                        <Bloque>.aux ⇐ ts.getBloqueActual();
                        ts.setBloqueActual(<Bloque>.b);
                }
                <ListaSentencias> }
                {
                        ts.setBloqueActual (<Bloque>.aux);
                        <Bloque>.salida ⇐ <Bloque>.b
                }
<ListaSentencias> → <Sentencia> {
```

```
                                    <ListaSentencias>.sentencia ⇐ <Sentencia>.salida;
                                    ts.getBloqueActual().agregarSentencia( <ListaSentencias>.sentencia)
                                    }
                                    <ListaSentencias> | λ

<Sentencia> → ; { <Sentencia>.salida ⇐ new SentenciaVacia() }
<Sentencia> → <Primario> ; { <Sentencia>.salida ⇐ new SentenciaLlamada( <Primario>.salida ,
                                                                        <Sentencia>.lexema) }|
                <Primario> <Asignacion> ;
                        { <Sentencia>.salida ⇐ <Asignacion>.salida}
<Sentencia> → <Tipo> {<ListaDecVars>.entrada ⇐ new Lista()}
                <ListaDecVars> ; {
                        <Sentencia>.listaVars ⇐ new lista();
                        for ( idMetVar var : <ListaDecVars>.salida)
                                Variable v = new Variable(var,"public", tipo.nombre);
                                <Sentencia>.listaVars.add(v);
                                ts.getBloqueActual().addVariable(v);
                                <Sentencia>.salida ⇐ new DecVars (<Tipo>.nombre,
                                                                  <Sentencia>.listaVars)
}
<Sentencia> → if (<Expresion>) <Sentencia> {
                                        <SentenciaElse>.expresion ⇐ <Expresion>.salida
                                        <SentenciaElse>.sentencia ⇐ <Sentencia>.salida
                                        }
                                    <SentenciaElse>
                                        { <Sentencia>.salida ⇐ <SentenciaElse> .salida }
<Sentencia> → while (<Expresion>) <Sentencia> {
                                    <Sentencia>.salida ⇐ new While (<Expresion>.salida ,
                                                                    sentencia.salida)
                                    }
<Sentencia> → <Bloque> { <Sentencia>.salida ⇐ <Bloque>.salida }
<Sentencia> → return <SentenciaP> ; { <Sentencia>.salida ⇐ new Return(<SentenciaP>.salida) }

<SentenciaElse> → else <Sentencia> { <SentenciaElse>.salida ⇐ new Else( <SentenciaElse>.expresion,
                                                                       <SentenciaElse>.sentencia ,
                                                                       <Sentencia>.salida) }|
                        λ { <SentenciaElse>.salida ⇐ new If
                                                ( <SentenciaElse>.expresion, SentenciaElse>.sentencia)
                        }
<SentenciaP> → <Expresion> { <SentenciaP>.salida ⇐ <Expresion>.salida}|
                    λ { <SentenciaP>.salida ⇐ null }
<Asignacion> → = <Expresion> { <Asignacion>.salida ⇐ new Asignacion
                                    ( <Expresion>.salida, <Asignacion>.Primario,
                                                                  <Asignacion>.Lexema) }|
                        λ { Asignacion.salida ⇐ null }
<Expresion> → <ExpAnd> { <ExprP>.entrada ⇐ <ExpAnd>.salida }
                <ExprP> { <Expresion>.salida ⇐ <ExprP>.salida }
<ExprP> → || <ExpAnd> { <ExpAnd>.entrada ⇐ <ExpAnd>.salida }
                <ExprP> { <ExprP>.salida ⇐ new ExpBinaria(<ExpP>.lexema,
                                    <ExprP>.Expresion, <ExpAnd>.salida) }|
                λ { <ExprP>.salida ⇐ <ExprP>.Expresion }
<ExpAnd> → <ExpIg> { <ExpAnd>.expresion ⇐ <ExpIg>.Salida }
```

<ExpAndP> { <ExpAnd>.salida ⇐ <ExpAndP>.salida }
<ExpAndP> → && <ExpIg> { <ExpAndP>.entrada ⇐ <ExpIg>.salida }
<ExpAndP> { <ExpAndP>.salida ⇐ new ExpBinaria(
<ExpP>.lexema, <ExprP>.Expresion, <ExpAnd>.salida) } |
λ { <ExpAndP>.salida ⇐ <ExpAndP>.Expresion }
<ExpIg> → <OpComp> { <ExpIg>.expresion ⇐ <OpComp>.Salida }
<ExpIgP> { <ExpIg>.salida ⇐ <ExpIgP>.salida }
<ExpIgP> → <OpIg> <Expcomp> { <ExpIgP>.entrada ⇐ <Expcomp>.salida }
<ExpIgP> { <ExpIgP>.salida ⇐ new ExpBinaria(
<ExpIgP>.lexema, <ExpIgP>.Expresion, <Expcomp>.salida) } |
λ { <ExpIgP>.salida ⇐ <ExpIgP>.Expresion }

<ExpComp> → <ExpAd> { <ExpComp>.expresion ⇐ <ExpAd>.Salida }
<ExpCompP> { <ExpComp>.salida ⇐ <ExpCompP>.salida }
<ExpCompP> → <OpComp> <ExpAd> { <ExpCompP>.entrada ⇐ <ExpAd>.salida
<ExpCompP>.salida ⇐ new ExpBinaria(
<ExpCompP>.lexema, <ExpCompP>.Expresion, <ExpAd>.salida) } |
instanceof { <ExpCompP>.entrada ⇐ new Literal(<ExpCompP>.lexema }
idClase {<ExpCompP>.salida ⇐ new ExpBinaria(
<ExpCompP>.lexema, <ExpCompP>.Expresion, <ExpAd>.salida) } |
λ {<ExpCompP>.salida ⇐ <ExpCompP>.Expresion}
<ExpAd> → <ExpMul> { <ExpAd>.expresion ⇐ <ExpMul>.Salida }
<ExpAdP> { <ExpAd>.salida ⇐ <ExpAdP>.salida }
<ExpAdP> → <OpAd> <ExpMul> { <ExpAdP>.entrada ⇐ <ExpMul>.salida }
<ExpAdP> { <ExpAdP>.salida ⇐ new ExpBinaria(
<ExpAdP>.lexema, <ExpAdP>.Expresion, <ExpAdP>.salida) } |
λ { <ExpAdP>.salida ⇐ <ExpAdP>.Expresion }
<ExpMul> → <ExprUn> { <ExpMul>.expresion ⇐ <ExprUn>.Salida }
<ExpMulP> { <ExpMul>.salida ⇐ <ExpMulP>.salida }
<ExpMulP> → <OpMul> <ExprUn> { <ExpMulP>.entrada ⇐ <ExprUn>.salida }
<ExpMulP> { <ExpMulP>.salida ⇐ new ExpBinaria(
<ExpMulP>.lexema, <ExpMulP>.Expresion, <ExpMulP>.salida) } |
λ { <ExpMulP>.salida ⇐ <ExpMulP>.Expresion }
<ExpCast> → [idClase] <Operando> { <ExpCast>.salida ⇐ new
ExpUnaria(<ExpCast>.lexema, <Operando>.salida }|
<Operando> { <ExpCast>.salida ⇐ <Operando>.salida }
<OpIg> → == | !=
<OpComp> → < | > | <= | >=
<OpAd> → + | -
<OpUn> → + | - | !
<OpMul> → * | / | %
<Operando> → <Literal> { <Operando>.salida ⇐ <Literal>.salida }|
<Primario> { <Operando>.salida ⇐ <Primario>.salida }
<Literal> → null {<Literal>.salida ⇐ new Literal(null.lexema)}|
true { <Literal>.salida ⇐ new Literal(true.lexema)}|
false { <Literal>.salida ⇐ new Literal(false.lexema)}|
intLiteral { <Literal>.salida ⇐ new Literal(intLiteral.lexema)}|
charLiteral { <Literal>.salida ⇐ new Literal(charLiteral.lexema)}|
stringLiteral {<Literal>.salida ⇐ new Literal(stringLiteral.lexema)}
<Primario> → (<PrimarioP>) <LlamadaoIdEnc> {<Primario>.salida ⇐ new
Literal(<PrimarioP>.salida, <LlamadaoIdEnc>.salida)}
<Primario> → this <LlamadaoIdEnc> {if (lexema = ".")

<Primario>.salida ⇐ new NodoThis(<Primario>.lexema,
<LlamadaoIdEnc>.salida)
else <Primario>.salida ⇐ new NodoThis(<Primario>.lexema)}
<Primario> → idMetVar <PrimarioPP> <LlamadaoIdEnc> {<Primario>.salida ⇐ <PrimarioPP>.salida}
<Primario> → new idClase <ArgsActuales> <LlamadaoIdEnc> {
<Primario>.salida ⇐ new Construir(IdClase.lexema,
<ArgsActuales>.salida ,<LlamadaoIdEnc>.salida) }
<PrimarioP> → <Expresion> {
<PrimarioP>.salida ⇐ new ExpConEncadenado(
<Expresion>.salida, <LlamadaoIdEnc>.salida) }|
idClase . <Llamada> {
<PrimarioP>.salida ⇐ new LlamadaConClase(
<PrimarioP>.lexema1, <PrimarioP>.lexema2,
<Llamada>.salida, <LlamadaoIdEnc>.salida) }
<PrimarioPP> → <ArgsActuales> {
<PrimarioPP>.salida ⇐ new Llamada(<PrimarioPP>.lexema,
<ArgsActuales>.salida, <LlamadaoIdEnc>.salida) }|
λ {
<PrimarioPP>.salida ⇐ new LlamadaVar(<PrimarioPP>.lexema,
<LlamadaoIdEnc>.salida) }
<LlamadaoIdEnc> → <LlamadaoIdEncadenado> <LlamadaoIdEnc> { <LlamadaoIdEncadenado>.salida ⇐
<LlamadaoIdEnc>.salida ;
<LlamadaoIdEnc>.salida ⇐ <LlamadaoIdEncadenado>.salida}|
λ {<LlamadaoIdEnc>.salida ⇐ null}

<LlamadaoIdEncadenado> → . idMetVar
{ <LlamadaoIdEncadenadoP> .entrada ⇐ idMetVar. Lexema}
<LlamadaoIdEncadenadoP>
{ llamadaIdEncadenado.salida ⇐ <LlamadaoIdEncadenadoP> .salida }
<LlamadaoIdEncadenadoP> → <ArgsActuales> {
<LlamadaoIdEncadenadoP>.salida ⇐ new LlamadaEncadenada(
<LlamadaoIdEncadenadoP>.entrada, null, argsActuales.salida) } |
λ { <LlamadaoIdEncadenadoP>.salida ⇐ new IdEncadenado(
<LlamadaoIdEncadenadoP>.entrada , null) }
<Llamada> → idMetVar <ArgsActuales> {<Llamada>.salida ⇐ <ArgsActuales>.salida }

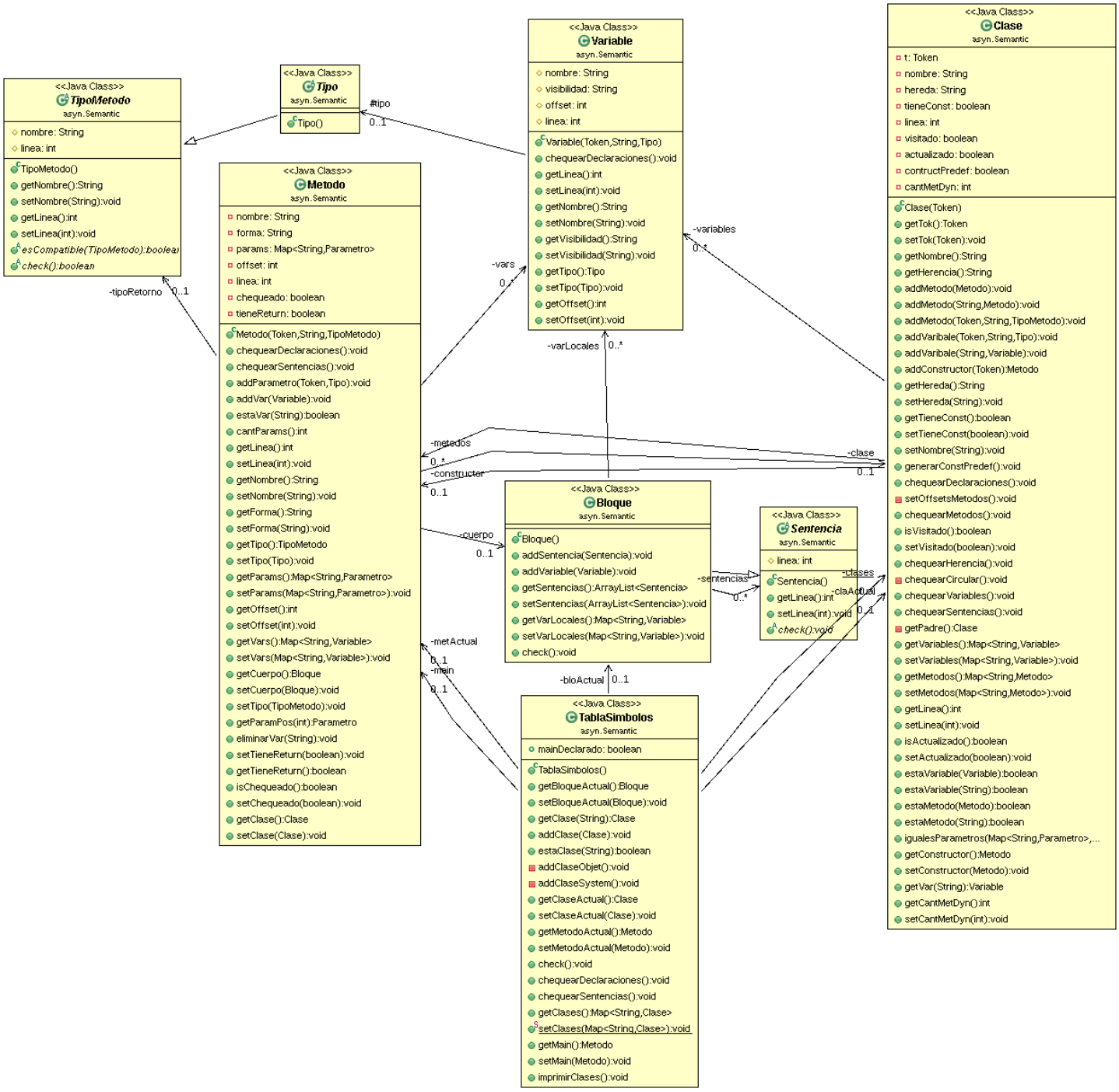
<ArgsActuales> → (<ArgsActualesP> {<ArgsActuales>.salida ⇐ <ArgsActualesP>.salida }
<ArgsActualesP> → <ListaExps>) { <ArgsActuales>.salida ⇐ <ListaExps>.salida }|
) {<ArgsActualesP>.salida ⇐ new Argumentos(null, null) }

<ListaExps> → <Expresion>{<ListaExpsP>.entrada ⇐ <Expresion>.salida}
<ListaExpsP> { <ListaExps>.salida ⇐ <ListaExpsP>.salida }
<ListaExpsP> → , <ListaExps> { <ListaExpsP>.salida ⇐ new Argumentos (ListaExpsP.entrada ,
listaExps.salida) }|
λ { <ListaExpsP>.salida ⇐ new Argumentos(<ListaExpsP>.entrada, null) }

Diagramas de Clases

Se presentará en primer lugar el diagrama de clases correspondiente a la TS, luego los diagramas correspondientes al AST y con aquellos diagramas con clases comunes tanto al AST como a la TS. Por última se mostrará un diagrama de clases completo, sin lo métodos.

1. Diagrama de la TS



Errores Semánticos Detectados

El compilador es capaz de detector los siguientes errores semánticos:

- No puede haber herencia circular.
- No pueden haber dos métodos con el mismo nombre.
- No pueden haber dos clases con el mismo nombre.
- No pueden declararse clases con nombre **Object** o **System**.
- En una clase no pueden haber dos variables de instancia con el mismo nombre.
- El nombre de una variable de instancia debe diferir del nombre de la clase y de los métodos.
- Los encabezados de los métodos (parámetros, tipo de retorno y variables locales) estén correctamente declarados.
- No pueden declararse métodos con el mismo nombre que la clase.
- No puede haber más de un constructor por clase.
- Un método/constructor no puede tener más de un parámetro (o variable local) con el mismo nombre, o con el nombre de una variable local (o parámetro) al mismo.
- Alguna clase debe tener un método llamado **main**.
- El método **main** debe ser estático.
- El método **main** no puede tener parámetros.
- Chequeo de herencia.
- Redefinición de métodos heredados (parámetros, retorno, forma).
- Tipo de la variable debe existir.
- Un método que no sea **void** debe poseer "**return**"
- El **return** debe coincidir con el tipo del método.
- El tipo de la expresión en la cláusula **if** es booleano
- Los tipos de las expresiones deben conformar.
- Las sentencias deben ser correctamente tipadas.
- Las asignaciones deben estar tipadas correctamente.
- Utilización de **this** en método estático.
- Llamada desde un método estático a uno dinámico.

Explicación de los Métodos de la Tabla de Simbolos

+getBloqueActual(): Devuelve la referencia al bloque que actualmente esta siendo analizado. Este bloque es representado con un atributo en la clase *TablaSimbolos*.

+setBloqueActual(Bloque): Cambia el valor de la variables que referencia al bloque que esta siendo analizado.

+getClase(String): Devuelve la referencia a la clase de la tabla de simbolos con el nombre pasado

+addClase(Clase): Agrega a la tabla de símbolos la clase pasada.

+estaClase(String): Se fija si la tabla contiene una clase con el nombre pasado y devulve verdadero o false según corresponda.

-addClaseObject(): Crea la clase *Object* y la agrega a la tabla de símbolos.

-addClaseSystem(): Crea la clase *System* y la agrega a la tabla de símbolos.

+getClaseActual(): Devuelve la referencia a la clase actual representada por un atributo de la clase *TablaSimbolos*.

+setClaseActual(Clase): Cambia el valor de la variable que representa a la clase actual con el valor pasado.

+getMetodoActual(): Devuelve el valor de la variable que representa y referencia al método que esta siendo analizado.

setMetodoActual(Metodo): Cambia el valor de la variable que representa al método actual con el valor pasado.

+check(): Chequea las declaraciones y las sentencias del proyecto que esta siendo analizado.

***+chequearDeclaraciones()*:** Chequea cada declaración de cada clase en el proyecto que esta siendo analizado.

***+chequearSentencias()*:** chequea cada sentencia de cada clase en el proyecto que esta siendo analizado.

+getClases(): Devuelve la referencia a la tabla de clases que contiene todas las clases del proyecto analizado.

+imprimirClases(): Imprime el nombre de cada clase existente en la tabla de símbolos con sus metodos y variables.

AST (Abstract Syntax Tree)

Cuando se procede a chequear los nodos del AST, de estos se controla que respeten los controles de sentencias.

Para ello, se chequea el bloque principal de cada método, y este a su vez, controla que toda la lista de sentencias que posee respeten los controles de sentencias.

Estas sentencias chequean que, según el tipo de sentencias que corresponda, estén correctas, y en caso de que la sentencia sea un bloque, se repite todo el proceso nuevamente.

Chequeos en Clase.java

Para el caso de los **chequeos de declaraciones**, en esta instancia se procede a controlar mediante distintas llamadas a métodos:

- Los métodos, se chequean sus declaraciones (haciendo uso de la clase *Metodo.java*) y se realizan distintos controles del método **main**.
- Que la herencia sea correcta, se realiza la herencia de los atributos y métodos del padre correspondiente a cada clase, si es que esta posee uno y se controla que no exista herencia circular.
- Por último, se procede a chequear que las variables estén bien instanciadas.

Para el caso de los **controles de sentencias**, se recorre todos los métodos de las clases, y se controla que cada uno respete los controles de sentencia.

Chequeos en Metodo.java

Cuando se realizan los controles de **declaraciones de un método**, se tiene en cuenta que:

- Se chequee que el tipo de retorno sea correcto.
- Todos los parámetros de la clase estén bien declarados (se realiza el chequeo de declaraciones y se agregarán los parámetros)

Para el caso del **chequeo de sentencias**, bastará con controlar que si tiene bloque, este respete los controles de sentencia.

Diseño: Analizador Semántico

Decisiones Generales

Para la implementación general, se crearon las estructuras necesarias para la realización de los chequeos semánticos (Representación Intermedia TS + ASTs). El módulo de Análisis Semántico que toma las estructuras y realiza todos los chequeos está dentro de una carpeta (*Semantic*).

4. Generación de Código CeIVM

En la siguiente sección se detallara brevemente la información para la realización de la última etapa, analizador completo + CeIVM.

Principalmente se contaran como se desarrolló parte del compilador, de acuerdo a lo brindado por la cátedra, tanto en explicaciones en clase como en el check point.

Control de Declaraciones

En cuanto a las clases, brindan el offset de las variables y el tamaño del CIR.

Con respecto a los métodos, se crea una etiqueta por cada método.

Control de Sentencias

- Se reparó el error en el que se volvían a chequear los métodos heredados en las clases, para evitar conflictos de tipo. Es decir, solo se chequea si el método es redefinido.
- La sentencia "*if*" se controló que se utilicen bien los Branches, BF al principio y JUMP al final del *then* o *else*.
- La sentencia "*while*", también se controló que se usen bien los *Branch*, de la misma forma que el *if*. Pero con etiquetas *FINWhile* y *PrincipioWhile*.
- La asignación se tuvo en cuenta que primero se chequee la expresión y luego el lado izquierdo. También se encontró una forma para saber cuándo una expresión Primaria se encuentra en el lado izquierdo, utilizando una variable "*esLadoIzq*" en todos los heredados de Primario.
- Las expresiones binarias: una vez que se chequearon las dos expresiones, se genera la instrucción al *token* operador del nodo.
- ***Unarias***: Luego de chequear la subexpresión, generan NEG para "-" o NOT para "!".
- ***Casting***: DUP, luego se carga el *IDClase* del objeto LOADREF 1, se carga *IDClase* de la clase a castear: PUSH *IDClase* y se comparan. Se genera también un mensaje de *error_casting*
- ***Instanceof***: *IDClase* del objeto se carga con LOADREF1, luego se carga *IDClase* a castear con PUSH *IDClase* y se comparan, generando EQ.
- Variable Primario: se pudo realizar tal como se propuso en el Check Point:

  Si es el caso que hay un lado izquierdo y no tiene encadenados:

  1. Si el id es un parámetro o variable local STORE n con su offset
  2. Si el id es una variable de instancia del objeto: LOAD 3, SWAP, STOREREFn con su offset

En los otros casos:

1. Si es un parámetro o variable local: LOAD
2. Para Variable de instancia de la clase actual: LOAD 3,

- *This*: Para traducir una expresión *this* solo es necesario cargar el *this* en la pila dado que a *this* solo se lo puede utilizar del lado izquierdo de una asignación si tiene un encadenado, sino tiene encadenado tiene que ser un lado derecho.
- Llamada Estática: Para traducir la llamada a un método estático primero hay que reservar un lugar para el retorno si el tipo de retorno del método es distinto de *Void*, para eso usamos un RMEM 1.
- Llamada a Constructor: implica crear el nuevo CIR antes de llamar al constructor.

Generación de Código

A continuación se mostrarán las secciones de código en las que se produjo la generación de código para finalmente obtener el código ejecutable de un archivo recibido como parámetro de entrada en el compilador, el mismo, por supuesto, fue escrito en código MiniJava.

GenCode.java

La clase fue definida en su totalidad para la generación de código.

```
public class GenCode {
	public static String path;
	private static GenCode gc;
	private int labelNbr = 0;
	private static int cantClases;
	private PrintWriter printWriter;
	public GenCode() {
	try {printWriter = new PrintWriter(new BufferedWriter( new FileWriter(path)));  }
		catch (Exception e) {e.printStackTrace();}
	}
	public static GenCode gen() {
		if(gc==null) {
			gc = new GenCode();
			cantClases=0;
		}
		return gc;
	}
	public void nl() {
		printWriter.println();
	}
	public void comment(String c) {
		printWriter.println("# " + c);
	}
	public void write(String c) {
		printWriter.println(c);
	}
	public void inicioUnidad() {
		printWriter.println("LOADFP # Guardo enlace dinamico");
		printWriter.println("LOADSP # Inicializo FP");
		printWriter.println("STOREFP");
		printWriter.println();
	}
	public void close() {
		printWriter.close();
	}
	public String genLabel() {
		return "l"+labelNbr++;
	}
	public static int contadorClase() {
		return cantClases++;
	}
}
```

Bloque.java

```
public void check() throws Exception {
                    .. . .
//FMEM DE LA CANT DE VARS LOCALES
                    GenCode.gen().write("FMEM "+varLocales.size()+" # Libero espacio de variables locales al bloque");
                    //CAMBIAR OFF VARS LOCALES
AnalizadorSyn.getTs().getMain().setOffVar(AnalizadorSyn.getTs().getMain().getOffVar()-varLocales.size());
. . .
}
```

Clase.java

```
public void check() {
                    String[] inst = print.split("\n");
                    for (String s : inst) {
                              GenCode.gen().write(s);
                    }
```

BloqueAux.java

```
public void chequearSentencias() throws Exception {
                    GenCode.gen().write("# Clase "+nombre);
                    GenCode.gen().write("# Creo la VTable");
                    GenCode.gen().nl();
                    GenCode.gen().nl();
                    GenCode.gen().write(".DATA");
                    String ls="DW ";
                    . . .
                    if(cant>0) {
                              ls = ls.substring(0,ls.length()-1); //Elimino la ultima coma
                              GenCode.gen().write("VT_"+nombre+": "+ls);
                    }
                    else {GenCode.gen().write("VT_"+nombre+": NOP");  }
                    GenCode.gen().nl();
                    GenCode.gen().nl();
                    GenCode.gen().write(".CODE");
                    . . .
          }
private void setOffsets() {
          . . .
                    for (Variable v : variables.values()) {
                              //cantAtrPadre++;
                              v.setOffset(++cantAtrPadre);
                    }
}
private void setOffsetsMetodos() {
                    . . .
                    for (String s : metodos.keySet()) {
                              if(s.charAt(0)!='@') {
                                        Metodo m = metodos.get(s);
                                        if(!m.getNombre().equals(nombre) && !m.getForma().equals("static")) {
                                                  if(!padre.estaMetodo(m)) {
                                                            m.setOffset(cantMetodosPadre++);
}         }         }         }         }
```

Construir.java

```
          public TipoMetodo check() throws Exception {
. . .
          GenCode.gen().write("RMEM 1 # Reservo lugar para el constructor de "+c.getNombre());
          GenCode.gen().write("PUSH "+(c.getVariables().size()+2)+" # Reservo lugar para variables de instancia y VT de la
clase " + c.getNombre());
                    GenCode.gen().write("PUSH lmalloc # Apilo la etiqueta del lmalloc");
                    GenCode.gen().write("CALL # Llamada al metodo malloc");
                    GenCode.gen().write("DUP");
          GenCode.gen().write("PUSH VT_"+c.getNombre()+" # Apilo direccion de la VTable de la clase "+c.getNombre());
          GenCode.gen().write("STOREREF 0 # Guardamos la Referencia a la VT en el CIR que creamos");
                    GenCode.gen().write("DUP");
                    Metodo m = c.getConstructor();
                    validarArgs(m);
                    if(enc != null) {
                              enc.setEsLadoIzq(this.esLadoIzq());
                              return enc.check(new TipoClase(tok));
                    }
                    GenCode.gen().write("PUSH "+m.getLabel()+" # Apilo etiqueta del constructor");
                    GenCode.gen().write("CALL # Llamo al constructor");
                    return new TipoClase(tok);
          }
private void validarArgs(Metodo m) throws Exception{
                    . . .
                              actual = actual.getArgs();
                              GenCode.gen().write("SWAP");
                    . . .
          }
```

DecVars.java

```
          public void check() throws Exception {
                    for (Variable v : vars) {
                    . . .
                              v.setOffset(m.getOffVar());
                              m.setOffVar(m.getOffVar()-1);
                              v.chequearDeclaraciones();
                    }
```

```
                //RMEM DE LA CANT EN VARS
                GenCode.gen().write("RMEM "+vars.size()+" # Reservo espacio para variables locales");

        }
```

Else.java

```
        public void check() throws Exception {
                . . .
                String finIf = "finIf"+GenCode.gen().genLabel();
                String lElse = "else"+GenCode.gen().genLabel();
                GenCode.gen().write("BF "+ lElse + " # Salto si la sentencia es falsa");
                sent.check();
GenCode.gen().write("JUMP "+finIf+" # Salto al fin del if para que no ejecute el else");
                GenCode.gen().write(lElse+": NOP # Codigo del else");
                sentElse.check();
                GenCode.gen().write(finIf+": NOP # Etiqueta fin if");
                GenCode.gen().nl();
        }
```

ExpBinaria.java

```
        public TipoMetodo check() throws Exception {
                TipoMetodo tipoEI = exp1.check();
                String op = tok.getLexema();
. . .                   switch(op) {
                      { case "+" :{GenCode.gen().write("ADD # Suma");break;}
                        case "-" :{         GenCode.gen().write("SUB # Resta");     break;}
                        case "*" :{         GenCode.gen().write("MUL # Multiplicacion");break;}
                        case "/" :{         GenCode.gen().write("Div # Division");break;}
                        case "%" :{         GenCode.gen().write("MOD # Modulo");break;}
                        }                           return new TipoInt(tok.getnLinea());
                }
                case "<" : case ">" : case "<=" : case ">=" :
                {if(!tipoED.getNombre().equals("int")||tipoEI.getNombre().equals("int")){
throw new Exception("Error, los tipos de las expresiones en la linea "+tok.getnLinea()+" no son compatibles");
                        }
                        switch(op) {
                        case "<" :{GenCode.gen().write("LT # Menor");break;}
                        case ">" :{GenCode.gen().write("GT # Mayor");break;}
                        case ">=" :{GenCode.gen().write("GE # Mayor o igual");break;}
                        case "<=" :{GenCode.gen().write("LE # Menor o igual");break;}
                        }
                        return new TipoBool(tok.getnLinea());
                }
                case "==" : case "!=" :
                {       if(!tipoED.esCompatible(tipoEI)) {
        throw new Exception("Error, los tipos de las expresiones en la linea "+tok.getnLinea()+" no son compatibles");
                        }
                        switch(op) {
                        case "==" :{GenCode.gen().write("EQ # Igual");break;}
                        case "!=" :{GenCode.gen().write("GT # Not Igual");break;}
                        }
                        return new TipoBool(tok.getnLinea());
                }
        case "instanceof":
                        . . . .
GenCode.gen().write("LOADREF 1 #Cargo el ID desde el CIR de la expresion analizada (izq)");
GenCode.gen().write("PUSH "+id+"  #Cargo el ID de la Clase (der)");
GenCode.gen().write("EQ #Comparo los IDs. El resultado queda en el tope de la pila");
                                return new TipoBool(tipoED.getLinea());
                        }
                default: //&& o ||
                {
        if(!tipoED.getNombre().equals("boolean") || !tipoEI.getNombre().equals("boolean")) {
        throw new Exception("Error, los tipos de las expresiones en la linea "+tok.getnLinea()+" no son compatibles");
                        }
                        switch(op) {
                        case "&&" :{GenCode.gen().write("AND # And");break;}
                        case "||" :{GenCode.gen().write("OR # Or");break;}
                        }
                        return new TipoBool(tok.getnLinea());
        }}      }
```

ExpUnaria.java

```
                public TipoMetodo check() throws Exception {
                TipoMetodo tipoExp = exp.check();
                switch (tok.getLexema()) {
                case "!": case "true": case "false":
                        if(!tipoExp.getNombre().equals("boolean")) {
throw new Exception("Error, el operador '!' no se puede aplicar a algo de tipo no booleano en linea "+tok.getnLinea());
                        }
                        GenCode.gen().write("NOT # Suma");
                        return new TipoBool(tok.getnLinea());

                case "+": case "-":
                        if(!tipoExp.getNombre().equals("int")) {
throw new Exception("Error, el operador unario no se puede aplicar a algo de tipo no entero en linea "+tok.getnLinea());
                        }
                        GenCode.gen().write("NEG # Suma");
                        return new TipoInt(tok.getnLinea());
                default: // CAST
                        TipoClase tipoCx = new TipoClase(tok);
                        if (!tipoCx.check())
```

```
                                        throw new Exception("Error, el tipo "+tok.getLexema()+" de la linea
"+tok.getnLinea()+" no existe.");
                                if (!tipoCx.esCompatible(tipoExp))
                                        throw new Exception("Error, el tipo "+tok.getLexema()+" no conforma con la
subexpresion en la linea. "+tok.getnLinea());
                        GenCode.gen().write("DUP  #Duplico la referencia al objeto");
                        GenCode.gen().write("LOADREF 1  #Cargo el ID desde el CIR de la expresion analizada (der)");
                        int id = TablaSimbolos.getClase(tok.getLexema()).getId();
                        GenCode.gen().write("PUSH "+id+"  #Cargo el ID de la clase a castear (izq)");
                        GenCode.gen().write("EQ  #Comparo los IDs");
                        GenCode.gen().write("BF error_casting  #Casteo incorrecto. Lanzo excepcion");
                                return tipoCx;

                        }
```

IdEncadenado.java

```
                        public TipoMetodo check(TipoMetodo tipo) throws Exception {
                                . . . .
                                        GenCode.gen().write("LOADREF "+v.getOffset()+" # Cargo variable de instancia
"+v.getNombre()+" de la clase "+c.getNombre());
                                }
                                else {
                                        throw new Exception("No se encontro la variable "+nombreVar+" que se intenta usar en
la linea "+tok.getnLinea());
                                }
                                return v.getTipo();
                        }
                        else { //Con encadenado

                                if(c.estaVariable(nombreVar)) {
                                        v = c.getVariables().get(nombreVar);
                                        if(c.getVar(nombreVar).getVisibilidad().equals("private")) {
                                                throw new Exception("No se puede acceder a la variable "+nombreVar+" en la
linea "+tok.getnLinea()+" ya que es "+v.getVisibilidad());
                                        }
                                        GenCode.gen().write("LOADREF "+v.getOffset()+" # Cargo variable de instancia
"+v.getNombre()+" de la clase "+c.getNombre());
                                        . . .
}
```

If.java

```
        public void check() throws Exception {
                //System.out.println(exp.check().getNombre());
                if(!exp.check().getNombre().equals("boolean")) {
                        throw new Exception("El tipo de la expresion en la clausula if en la linea
"+exp.getToken().getnLinea()+" no es boolean");
                }
                String finIf = "finIf"+GenCode.gen().genLabel();
                GenCode.gen().write("BF "+ finIf + " # Salto si la sentencia es falsa");
                sent.check();
                GenCode.gen().write(finIf+": NOP # Etiqueta fin if");
                GenCode.gen().nl();
        }
```

Literal.java

```
        public Tipo check() throws Exception{
                Tipo aux=null;
                switch(tok.getName()) {
                        case "null" : {aux = new TipoClase(tok); break;}
                        case "true" : { aux = new TipoBool(tok.getnLinea()); break;}
                        case "false" :      { aux = new TipoBool(tok.getnLinea()); break;}
                        case "intLiteral" : { aux = new TipoInt(tok.getnLinea()); break;}
                        case "charLiteral" : { aux = new TipoChar(tok.getnLinea()); break;}
                        case "stringLiteral" :        { aux = new TipoString(tok.getnLinea()); break;}
                        default : return null;
                }
                aux.gen(tok.getLexema());
                return aux;
        }
```

Llamada.java

```
public TipoMetodo check() throws Exception {

          .  .   . .   .
                        if(!m.getTipo().getNombre().equals("void")) {
                                GenCode.gen().write("RMEM 1 # Reservo lugar para el retorno del metodo");
                                if(m.getForma().equals("dynamic")) {
                                        GenCode.gen().write("SWAP");
                                }
                        }
validarArgs(m);
if(m.getForma().equals("static")) {
                                GenCode.gen().write("PUSH "+m.getLabel()+" # Apilo la etiqueta del metodo");
                                GenCode.gen().write("CALL # Llamo al metodo");
                        }
                        else {
                                GenCode.gen().write("LOAD 3 # Cargo THIS");
                                GenCode.gen().write("DUP");
                                GenCode.gen().write("LOADREF 0 # Cargo la VTable");
                                GenCode.gen().write("LOADREF "+m.getOffset()+" # Cargo el metodo "+m.getNombre());
                                GenCode.gen().write("CALL # Llamo al metodo");
                        }
```

```
		. . .
	}
```

LlamadaConClase.java

```
public TipoMetodo check() throws Exception {
		.	.	.	.	.
	if(!m.getTipo().getNombre().equals("void"))
			GenCode.gen().write("RMEM 1 # Reservo memoria para el retorno del metodo");
		GenCode.gen().write("PUSH "+m.getLabel()+" # Apilo la etiqueta del metodo");
		GenCode.gen().write("CALL # Llamo al metodo");
.	.	.	.	.
	}
```

LlamadaEncadenada.java

```
public TipoMetodo check(TipoMetodo tipo) throws Exception {
			.	.	.	.
if(!m.getTipo().getNombre().equals("void")) {
		GenCode.gen().write("RMEM 1 # Reservo lugar para el retorno del metodo");
					if(m.getForma().equals("dynamic")) {
						GenCode.gen().write("SWAP");
					}
				}
				if(m.getForma().equals("static")) {
					GenCode.gen().write("POP");
				}
				validarArgs(m);

	if(m.getForma().equals("static")) {
		GenCode.gen().write("PUSH "+m.getLabel()+" # Apilo la etiqueta del metodo");
		GenCode.gen().write("CALL # Llamo al metodo");
				}
	else {
	GenCode.gen().write("DUP");
	GenCode.gen().write("LOADREF 0 # Cargo la VTable");
	GenCode.gen().write("LOADREF "+m.getOffset()+" # Cargo el metodo "+m.getNombre());
	GenCode.gen().write("CALL # Llamo al metodo");
	}
	. . . . .
	}
```

LlamadaVar.java

```
	public TipoMetodo check() throws Exception {
		. . .
		if (esLadoIzq() && enc==null){
			if (m.estaVar(nombreVar) || m.getParams().containsKey(nombreVar))
				GenCode.gen().write("STORE "+v.getOffset()+" # ");
			else {
				// var instancia
				if(c.estaVariable(nombreVar)) {
					if(!m.getForma().equals("static")) {
						v = c.getVar(nombreVar);
						GenCode.gen().write("LOAD 3 # Cargo This");
						GenCode.gen().write("SWAP");
	GenCode.gen().write("STOREREF "+v.getOffset()+" # Cargo el valor de la variable");
					} else
						throw new Exception("Se intento referenciar a una variable de
instancia en la linea "+tok.getnLinea()+" desde un metodo estatico");
				} else
					throw new Exception("La variable "+nombreVar+" en la linea
"+tok.getnLinea()+" no esta definida");
			}
		} else {
			if (m.estaVar(nombreVar) || m.getParams().containsKey(nombreVar))
				GenCode.gen().write("LOAD "+v.getOffset()+" # ");
			else {
				// var instancia
				if(c.estaVariable(nombreVar)) {
					if(!m.getForma().equals("static")) {
						v = c.getVar(nombreVar);
						GenCode.gen().write("LOAD 3 # Cargo This");
						GenCode.gen().write("SWAP");
						GenCode.gen().write("LOADREF "+v.getOffset()+" # Cargo el valor
de la variable");
					}
		. . .
}
```

Metodo.java

```
	public void chequearDeclaraciones() throws Exception{
		.	.	.	.	.	.
		for (Parametro p : params.values()) {
			p.chequearDeclaraciones();
			if(forma.equals("static") && !nombre.equals(clase.getNombre())) {
				p.setOffset(2 + cantParams() - p.getPosicion()); //3+cantParams-nroParam
			}
			else {
				p.setOffset(3 + cantParams() - p.getPosicion());
			}
			//Agrego a los parametros como variables locales
			vars.put(p.getNombre(), p);
		}
	}
```

```
	public void chequearSentencias() throws Exception{
				. . . . . .
			if(nombre.equals(clase.getNombre())) { //Es constructor
				GenCode.gen().write(getLabel()+": NOP # CONSTRUCTOR "+nombre);
			}
			else {GenCode.gen().write(getLabel()+": NOP # METODO "+nombre);}
		GenCode.gen().inicioUnidad();
		setOffVar(0);
		if(cuerpo != null) {
			cuerpo.check();
		}
		GenCode.gen().write("STOREFP # Restablezco el contexto");

		if(forma.equals("static") && !nombre.equals(clase.getNombre())) {
			//Si es estatico y no es el constructor
			GenCode.gen().write("RET "+params.size()+" # Retorno y libero espacio de los parametros del
metodo "+nombre);
		}
		else { //Si es dinamico o es constructor
			GenCode.gen().write("RET "+(params.size()+1)+" # Retorno y libero espacio de los parametros
del metodo y del THIS "+nombre);
		}
		GenCode.gen().nl();
	}
```

NodoThis.java

```
	public TipoMetodo check() throws Exception {
	.	.	.	.
		GenCode.gen().write("LOAD 3 # Cargo el THIS");
	.	.	.	.
	}
```

Return.java

```
	public void check() throws Exception {
		.	.	.
	if(m.getForma().equals("static")) {
GenCode.gen().write("STORE "+(3+m.getParams().size())+" # Guardo valor de retorno del metodo "+m.getNombre());
	}
else {
GenCode.gen().write("STORE "+(4+m.getParams().size())+" # Guardo valor de retorno del metodo "+m.getNombre());
			}
if(m.getVars().size()-m.getParams().size() > 0) {
	GenCode.gen().write("FMEM "+(m.getVars().size()-m.getParams().size())+" # Libero espacio de variable locales al
metodo "+m.getNombre());
			}
GenCode.gen().write("STOREFP");

if(m.getForma().equals("static")) { //Si es estatico
	GenCode.gen().write("RET "+m.getParams().size()+" # Retorno y libero espacio de los parametros del metodo
"+m.getNombre());
			}
else { //Si es dinamico
	GenCode.gen().write("RET "+(m.getParams().size()+1)+" # Retorno y libero espacio de los parametros del metodo y
del THIS "+m.getNombre());
		}   }
	.	.	.	.
	}
```

SentenciaLlamada.java

```
public void check() throws Exception{
		TipoMetodo tm = p.check();
		if(!tm.getNombre().equals("void")) {
			GenCode.gen().write("POP");
		}
	}
```

TablaSimbolos.java

```
private void addClaseSystem() throws Exception{
		....
		m.setClase(c);
		m.setOffset(0);
		. . .
		m.setOffset(1);
		String printB = "LOAD 3\nBPRINT";
		m.setCuerpo(new BloqueAux(printB));
		m.setOffParam(1);
		c.addMetodo(m);
	m = new Metodo(new Token("idMetVar","printC",0), "static", new TipoVoid(),claActual);
		m.addParametro(new Token("","c",0), new TipoChar(0));
		m.setClase(c);
		m.setOffset(2);
		String printC = "LOAD 3\nCPRINT";
		m.setCuerpo(new BloqueAux(printC));
		m.setOffParam(1);
		c.addMetodo(m);
	m = new Metodo(new Token("idMetVar","printI",0), "static", new TipoVoid(),claActual);
		m.addParametro(new Token("","i",0), new TipoInt(0));
		m.setClase(c);
		m.setOffset(3);
		String printI = "LOAD 3\nIPRINT";
		m.setCuerpo(new BloqueAux(printI));
		m.setOffParam(1);
```

```
                c.addMetodo(m);
        m = new Metodo(new Token("idMetVar","printS",0), "static", new TipoVoid(),claActual);
                m.addParametro(new Token("","s",0), new TipoString(0));
                m.setClase(c);
                m.setOffset(4);
                String printS = "LOAD 3\nSPRINT";
                m.setCuerpo(new BloqueAux(printS));
                m.setOffParam(1);
                c.addMetodo(m);
        m = new Metodo(new Token("idMetVar","println",0), "static", new TipoVoid(),claActual);
                m.setClase(c);
                m.setOffset(5);
                String println = "PRNLN";
                m.setCuerpo(new BloqueAux(println));
                m.setOffParam(0);
                c.addMetodo(m);
        m = new Metodo(new Token("idMetVar","printBln",0), "static", new TipoVoid(),claActual);
                m.addParametro(new Token("","b",0), new TipoBool(0));
                m.setClase(c);
                m.setOffset(6);
                String printBln = printB + '\n' + println;
                m.setCuerpo(new BloqueAux(printBln));
                m.setOffParam(1);
                c.addMetodo(m);
        m = new Metodo(new Token("idMetVar","printCln",0), "static", new TipoVoid(),claActual);
                m.addParametro(new Token("","c",0), new TipoChar(0));
                m.setClase(c);
                m.setOffset(7);
                String printCln = printC + '\n' + println;
                m.setCuerpo(new BloqueAux(printCln));
                m.setOffParam(1);
                c.addMetodo(m);
        m = new Metodo(new Token("idMetVar","printIln",0), "static", new TipoVoid(),claActual);
                m.addParametro(new Token("","i",0), new TipoInt(0));
                m.setClase(c);
                m.setOffset(8);
                String printIln = printI + '\n' + println;
                m.setCuerpo(new BloqueAux(printIln));
                m.setOffParam(1);
                c.addMetodo(m);
        m = new Metodo(new Token("idMetVar","printSln",0), "static", new TipoVoid(),claActual);
                m.addParametro(new Token("","s",0), new TipoString(0));
                m.setClase(c);
                m.setOffset(9);
                String printSln = printS + '\n' + println;
                m.setCuerpo(new BloqueAux(printSln));
                m.setOffParam(1);
                c.addMetodo(m);
. . . }
        public void chequearSentencias() throws Exception{
                GenCode.gen().comment("<<<<< Inicializacion >>>>");
                GenCode.gen().nl();
                GenCode.gen().write(".CODE");
                GenCode.gen().write("PUSH lheap_init");
                GenCode.gen().write("CALL");
                GenCode.gen().write("PUSH "+ main.getLabel()+" # Metodo Main");
                GenCode.gen().write("CALL");
                GenCode.gen().write("HALT");
                GenCode.gen().nl();
                GenCode.gen().nl();
                GenCode.gen().write("lmalloc: LOADFP # Inicialización unidad");
                GenCode.gen().write("LOADSP");
                GenCode.gen().write("STOREFP # Finaliza inicialización del RA");
                GenCode.gen().write("LOADHL # hl");
                GenCode.gen().write("DUP # hl");
                GenCode.gen().write("PUSH 1 # 1");
                GenCode.gen().write("ADD # hl + 1");
                GenCode.gen().write("STORE 4 # Guarda resultado (puntero a base del bloque)");
                GenCode.gen().write("LOAD 3 # Carga cantidad de celdas a alojar (par´ametro)");
                GenCode.gen().write("ADD");
                GenCode.gen().write("STOREHL # Mueve el heap limit (hl)");
                GenCode.gen().write("STOREFP");
                GenCode.gen().write("RET 1 # Retorna eliminando el parámetro");
                GenCode.gen().nl();
                GenCode.gen().nl();
                GenCode.gen().write("lheap_init: RET 0 # Inicialización simplificada del .heap");
                GenCode.gen().nl();
                GenCode.gen().nl();
                GenCode.gen().write("# ---------GCI DEL CODIGO FUENTE---------------");
                GenCode.gen().write(".DATA");
                GenCode.gen().write("VT_Object: NOP");
                GenCode.gen().write("Object_Object: NOP");
                GenCode.gen().write("mensaje_error_casting: DW \"[Excepcion] Se produjo error al intentar castear\",0
                #Mensaje para casteos erroneos");
                GenCode.gen().write(".CODE");
                GenCode.gen().write("error_casting: PUSH mensaje_error_casting #Apilo msj de error");
                GenCode.gen().write("SPRINT");
                GenCode.gen().write("HALT");
                for (Clase c : clases.values()) {
                        if(!c.getNombre().equals("Object")) {claActual = c;
                                                        c.chequearSentencias();
                        }
```

```
                }
                GenCode.gen().close();
        }
```

TipoBool.java

```
        public void gen(String s) {
                if (s.equals("true"))
                        GenCode.gen().write("PUSH 1 # Apilo el booleano true");
                else
                        GenCode.gen().write("PUSH 0 # Apilo el booleano false");
        }
```

TipoChar.java

```
public void gen(String s) {GenCode.gen().write("PUSH "+ (int) s.charAt(0)+" # Apilo el caracter "+s.charAt(0));}
```

TipoClase.java

```
public void gen(String s) {}
```

TipoInt.java

```
        public void gen(String s) {GenCode.gen().write("PUSH "+s+" # Apilo el valor "+s);}
```

TipoMetodo.java

```
public abstract void gen(String s);
```

TipoString.java

```
        public void gen(String s) {
                GenCode.gen().write(".DATA");
                String et = GenCode.gen().genLabel();
                GenCode.gen().write("l_str"+et+": DW \""+s+"\",0");
                GenCode.gen().write(".CODE");
                GenCode.gen().write("PUSH l_str"+et+" # Apilo etiqueta del String");
        }
```

TipoVoid.java

```
        public void gen(String s) {}
```

Variable.java

```
                public int getOffset() { return offset; }
                public void setOffset(int offset) {this.offset = offset;}
```

While.java

```
public void check() throws Exception {
                String finWhile = "finWhile"+GenCode.gen().genLabel();
                String lWhile = "while"+GenCode.gen().genLabel();
                GenCode.gen().write(lWhile+": NOP # Etiqueta while");
                TipoMetodo tipoExp = exp.check();
                if(!tipoExp.getNombre().equals("boolean")) {
throw new Exception("La expresion en la sentencia while de la linea "+exp.getToken().getnLinea()+" no es booleana");
                }
GenCode.gen().write("BF "+ finWhile + " # Si es falso salgo del bucle");
                sent.check();
                GenCode.gen().write("JUMP "+lWhile+" # Salto al label del while");
                GenCode.gen().write(finWhile+": NOP # Etiqueta finWhile");
        }
```