Compilador Minijava

Manual Técnico

Compiladores e Intérpretes 2014

Brenda S. Dilschneider LU: 92774

Francisco Cuenca LU: 94294

Índice

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.

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.
Cada lexema Minijava corresponde a:

- Palabras Claves
- Nombres de tipos primitivos
- Literales
- Símbolos de puntuación
- Operadores
- Identificadores

Alfabeto de entrada

El alfabeto de entrada ∑ se corresponde con todos los caracteres del código ASCII extendido. Ya que consideramos que se reconocerá cualquier caracter.
Símbolos del código ASCII extendido.

Caracteres ASCII de control

00	NULL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	HT	(tab horizontal)
10	LF	(nueva línea)
11	VT	(tab vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc.vínculo datos)
17	DC1	(control disp. 1)
18	DC2	(control disp. 2)
19	DC3	(control disp. 3)
20	DC4	(control disp. 4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sínc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin del medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

ASCII extendido (Página de código 437)

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Tokens reconocidos

letra={a|b|..|z|A|B|..|Z}
digito={0|1|2|3|4|5|6|7|8|9}

Nombre	Expresión regular	Descripción
id	(_ + letra)(letra+digito+_)*	Identificador
number	digito(digito)*	Literal entero
string	"∑*-{\n}"	Literal string
booleanLiteral	false	Literal booleano
booleanLiteral	true	Literal booleano
nullLiteral	null	Literal nulo
character	'∑-{\n,\t}' | '\∑-{n,t}'	Literal caracter
openBrace	(	Puntuación
closeBrace	)	Puntuación
openCB	{	Puntuación
closeCB	}	Puntuación
semicolon	;	Puntuación
comma	,	Puntuación
dot	.	Puntuación
opGT	>	Operador
opGTEQ	>=	Operador
opLT	<	Operador
opLTEQ	<=	Operador
opNEG	!	Operador
opNEQ	!=	Operador
opEQ	==	Operador

opADD	+	Operador
opMULT	*	Operador
opSUB	-	Operador
opAND	&&	Operador
opOR	||	Operador
opMOD	%	Operador
ASSIG	=	Asignación
class	class	Keyword
extends	extends	Keyword
varinst	varinst	Keyword
Varlocal	Varlocal	Keyword
static	static	Keyword
dynamic	dynamic	Keyword
void	void	Keyword
Boolean	Boolean	Keyword
char	char	Keyword
int	int	Keyword
String	String	Keyword
if	if	Keyword
else	else	Keyword
while	while	Keyword
for	for	Keyword
return	return	Keyword
this	this	Keyword
new	new	Keyword

Errores Léxicos

El analizador léxico es capaz de detectar los siguientes errores.

Nombre del error	Motivo
Invalid Character Constant	Un literal carácter mal formado.
Invalid Comment	Un comentario mal formado.
Invalid String Literal	Un Literal String mal formado.
Invalid Character	Se encuentra un carácter que no corresponde al comienzo de ningún Token.
Invalid Number Literal	Un Número mal formado. Se considera un numero mal formado si la siguiente expresión regular se cumple: digito (digito)*(letra | (| { | " | ' | .) ejemplos: 123A 123(123{

Gramática modificada

Al momento de realizar el analizador léxico se nos requirió modificar la gramática presentada por la cátedra para que se preserven las reglas de precedencia y asociatividad.
La gramática resultante fue la siguiente:
<Expresion> → <Expresion> || <E5> | <E5>
<E5> → <E5> && <E4> | <E4>
<E4> → <E4> != <E3> | <E4> == <E3> | <E3>
<E3> → <E3> < <E2> | <E3> <= <E2> | <E3> > <E2> | <E3> >= <E2> | <E2>
<E2> → <E2> - <E1> | <E2> + <E1> | <E1>
<E1> → <E1>*<ExprUnaria> |<E1>/<ExprUnaria> |<E1>%<ExprUnaria> |<ExprUnaria>
<ExprUnaria>→ + <ExprUnaria> | -<ExprUnaria> | ! <ExprUnaria> | <Primario>

Clases utilizadas

- LexicalAnalyzer: Implementa un Autómata Finito No Determinista para transformar el código en tokens y detecta errores correspondientes a esta etapa.
- Principal: Obtiene la lista de tokens.
- Token: Estructura de un token (nombre, lexema, línea).
- LexicalException: Representa un error de tipo léxico

Decisiones de Diseño

Los comentarios son presentados de la forma estipulada en la descripción de la sintaxis de Minijava, es decir:

- Comentarios Simples: "//"
- Comentarios Multi-Linea : "/* ..*/"

Los errores son *siempre* mostrados por pantalla, en caso de encontrar un error se almacenan los tokens reconocidos hasta el momento.

Las palabras reservadas (*alias keywoards*) son almacenadas en una estructura ArrayList implementada por Java.
El autómata es implementado en *LexicalAnalyzer* mediante una estructura switch para brindar mayor facilidad de lectura y eficiencia.
Por último, la cadena vacía "" es reconocida como un literal String por lo tanto su salida deberá ser mostrada como:

```
Nro de linea    Token           Lexema
        1       stringLiteral
Compilacion exitosa.
```

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.

Gramática

Eliminación de elementos de la gramática EBNF

<Inicial> → <Clase> <Inicial> | <Clase>
<Clase> → **class** identificador <Herencia> { <MiembroL> } | **class** identificador { <MiembroL> }
<MiembroL> → <Miembro> <MiembroL> | λ
<Herencia> → **extends** identificador
<Miembro> → <Atributo> | <Ctor> | <Metodo>
<Atributo> → **varinst** <Tipo> <ListaDecVars> **;**
<Metodo> → <ModMetodo> <TipoMetodo> identificador <ArgsFormales> <Bloque>
<Ctor> → identificador <ArgsFormales> <Bloque>
<ArgsFormales> → **(**<ListaArgsFormales>**)** | **()**
<ListaArgsFormales> → <ArgFormal> | <ArgFormal> , <ListaArgsFormales>
<ArgFormal> → <Tipo> identificador
<ModMetodo> → **static** | **dynamic**
<TipoMetodo> → <Tipo> | **void**
<Tipo> → <TipoPrimitivo > | identificador
<TipoPrimitivo> → **boolean** | **char** | **int** | **String**
<ListaDecVars> → identificador | identificador , <ListaDecVars>
<Bloque> → **{** <SentenciaL> **}**
<SentenciaL> → <Sentencia><SentenciaL> | λ
<Sentencia> → **;**
<Sentencia> → <Asignacion>
<Sentencia> → <SentenciaSimple> **;**
<Sentencia> → **varlocal** <Tipo> <ListaDecVars> ;
<Sentencia> → **if (**<Expresion>**)** <Sentencia>
<Sentencia> → **if (**<Expresion>**)** <Sentencia> **else** <Sentencia>
<Sentencia> → **while (**<Expresion>**)** <Sentencia>
<Sentencia> → **for (**<Asignacion> **;** <Expresion> **;** <Asignacion>**)** <Sentencia>
<Sentencia> → <Bloque>
<Sentencia> → **return** <Expresion> **;** | **return ;**
<Asignacion> → <LadoIzquierdo> = <Expresion>
<LadoIzquierdo> → identificador | identificador . <LadoIzquierdo>
<SentenciaSimple> → **(**<Expresion>**)**
<Expresion> → <Expresion> **||** <Expr5> | <Expr5>

<Expr5> → <Expr5> **&&** <Expr4> | <Expr4>
<Expr4> → <Expr4> **==** <Expr3> | <Expr4> **!=** <Expr3> | <Expr3>
<Expr3> → <Expr3> **>=** <Expr2> | <Expr3> **<=** <Expr2> | <Expr3> **<** <Expr2> | <Expr3> **>** <Expr2> | <Expr2>
<Expr2> → <Expr2> - <Expr1> | <Expr2> + <Expr1> | <Expr1>
<Expr1> → <Expr1> * <ExprUnaria>| <Expr1> / <ExprUnaria>| <Expr1> % <ExprUnaria>| <ExprUnaria>
<ExprUnaria>→ + <ExprUnaria> | - <ExprUnaria> | ! <ExprUnaria> | <Primario>
<Primario> → **this**
<Primario> → <Literal>
<Primario> → **(** <Expresion> **)** <LlamadaL>
<Primario> → identificador <LlamadaL>
<Primario> → **new** identificador <ArgsActuales> <LlamadaL>
<Primario> → identificador <ArgsActuales> <LlamadaL>
<LlamadaL> → <Llamada><LlamadaL> | λ
<Llamada> → **.**identificador <ArgsActuales> | .identificador
<Literal> → **null** | **true** | **false** | **intLiteral** | **charLiteral** | **stringLiteral**
<ArgsActuales> → (<ListaExps>) | ()
<ListaExps> → <Expresion> | <Expresion> , <ListaExps>

Eliminación de recursión a izquierda

<Inicial> → <Clase> <Inicial> | <Clase>
<Clase> → **class** identificador <Herencia> **{** <MiembroL> **}** | **class** identificador **{** <MiembroL> **}**
<MiembroL> → <Miembro> <MiembroL> | λ
<Herencia> → **extends** identificador
<Miembro> → <Atributo> | <Ctor> | <Metodo>
<Atributo> → **varinst** <Tipo> <ListaDecVars> **;**
<Metodo> → <Modmetodo> <TipoMetodo> **identificador** <ArgsFormales> <Bloque>
<Ctor> → **identificador** <ArgsFormales> <Bloque>
<ArgsFormales> → (<ListaArgsFormales>) | ()
<ListaArgsFormales> → <ArgFormal> | <ArgFormal> , <ListaArgsFormales>
<ArgFormal> → <Tipo> **identificador**
<ModMetodo> → **static** | **dynamic**
<TipoMetodo> → <Tipo> | **void**
<Tipo> → <TipoPrimitivo> | **identificador**
<TipoPrimitivo> → **boolean** | **char** | **int** | **string**
<ListaDecVars> → **identificador** | **identificador ,** <ListaDecVars>
<Bloque> → {<SentenciaL>}
<SentenciaL> → <Sentencia> <SentenciaL> |λ
<Sentencia> → ;
<Sentencia> → <Asignación>;
<Sentencia> → <SentenciaSimple>;
<Sentencia> → varlocal <Tipo> <ListaDecVars>;

<Sentencia> → **if** (<Expresion>) <Sentencia>
<Sentencia> → **if** (<Expresion>) <Sentencia> **else** <Sentencia>
<Sentencia> → **while** (<Expresion>) <Sentencia>
<Sentencia> → **for** (<Asignación> ; <Expresión>; <Asignación>) | <Sentencia>
<Sentencia> → <Bloque>
<Sentencia> → **return** <Expresion> ; | **return** ;
<Asignacion>→<LadoIzquierdo> = <Expresion>
<LadoIzquierdo> → **identificador** | **identificador .** <LadoIzquierdo>
<SentenciaSimple> → (<Expresion>)
<Expresion> → <Expr5> <ExprP>
<ExprP> → **||** <Expr5> <ExprP> | λ
<Expr5> → <Expr4> <Expr5P>
<Expr5P> → **&&** <Expr4><Expr5P> |λ
<Expr4> → <Expr3> <Expr4P>
<Expr4P> → **==** <Expr3> <Expr4P> | **!=** <Expr3> <Expr4P> |λ
<Expr3> → <Expr2> **>=** <Expr2> | <Expr2> **<=** <Expr2> | <Expr2> **<** <Expr2> | <Expr2> **>** <Expr2> |
<Expr2>
<Expr2> → <Expr1> <Expr2P>
<Expr2P> → <
Expr1> <Expr2P> | **+** <Expr1> <Expr2P> |λ
<Expr1> → <ExprUnaria> <Expr1P>
<Expr1P> → *<ExprUnaria> <Expr1P> | /<ExprUnaria> <Expr1P> | %<ExprUnaria> <Expr1P>| λ
<ExprUnaria> → + <ExprUnaria> | <
ExprUnaria> | ! <ExprUnaria> | <Primario>
<Primario> → **this**
<Primario> → <Literal>
<Primario> → **(** <Expresion> **)** <LlamadaL>
<Primario> → **identificador** <LlamadaL>
<Primario> → **new identificador** <ArgsActuales> <LlamadaL>
<Primario> → i**dentificador** <ArgsActuales> <LlamadaL>
<LlamadaL> → <Llamada><LlamadaL> |λ
<Llamada> → **.**identificador <ArgsActuales>
<Literal> → **null** | **true** | **false** | **intLiteral** | **charLiteral** | **stringLiteral**
<ArgsActuales> → **(** <ListaExps> **)** | **()**
<ListaExps> → <Expresion> | <Expresion> **,** <ListaExps>

Factorización

<Inicial> → <Clase> <InicialP>
<InicialP> → <Inicial> | λ
<Clase> → **class identificador** <ClaseP>
<ClaseP> → <Herencia> { <MiembroL>} | {<MiembroL>}
<MiembroL> → <Miembro> <MiembroL> | λ
<Herencia> → **extends identificador**
<Mmiembro> → <Atributo>| <Ctor>| <Metodo>
<Atributo> → **varInst** <Tipo> <ListaDecVars> ;
<Metodo> → <ModMetodo> <TipoMetodo> **identificador** <ArgsFormales> <Bloque>

<Ctor> → **identificador** <ArgsFormales> <Bloques>
<ArgsFormales> → (<ArgsFormalesP>
<ArgsFormalesP> → <ListaArgsFormales>) |)
<ListaArgsFormales> → <ArgFormal> <ListaArgsFormalesP>
<ListaArgsFormalesP> → , <ListaArgsFormales> |λ
<ArgFormal> → <Tipo> **identificador**
<ModMetodo> → **static** | **dynamic**
<TipoMetodo> → <Tipo> | **void**
<Tipo> → <TipoPrimitivo> | **identificador**
<TipoPrimitivo> → **boolean** | **char** | **int** | **String**
<ListaDecVars> → **identificador** <ListaDecVarsP>
<ListaDecVarsP> → , <ListaDecVars> |λ
<Bloque> → {<SentenciaL>}
<SentenciaL> → <Sentencia> <SentenciaL> |λ
<Sentencia> → ;
<Sentencia> → <Asignacion>
<Sentencia> → <SentenciaSimple>;
<Sentencia> → **varLocal** <Tipo><ListaDecVars>;
<Sentencia> → **if** (<Expresion>) <Sentencia> <SentenciaP>
<SentenciaP> → **else** <Sentencia> | λ
<Sentencia> → **while** (<Expresion>) <Sentencia>
<Sentencia> → **for** (<Asignacion> ; <Expresion> ; <Asignacion>) <Sentencia>
<Sentencia> → <Bloque>
<Sentencia> → **return** <SentenciaPP>
<SentenciaPP> → <Expresion> ; | ;
<Asignacion> → <LadoIzquierdo> = <Expresion>
<LadoIzquierdo> → **identificador** <IdEncadenados> | **identificador .** <LadoIzquierdo>
<IdEncadenados> → λ
<IdEncadenados> → **. identificador** <IdEncadenados>
<SentenciaSimple> → (<Expresion>)
<Expresion> → <Expr5> <ExprP>
<ExprP> → || <Expr5> <ExprP> | λ
<Expr5> → <Expr4> <Expr5P>
<Expr5P> → && <Expr4> <Expr5P> |λ
<Expr4> → <Expr3> <Expr4P>
<Expr4P> → == <Expr3> <Expr4P> | != <Expr3> <Expr4P> | λ
<Expr3> → <Expr2> <Expr3P>
<Expr3P> → >= <Expr2> || <= <Expr2> || > <Expr2> || < <Expr2> || λ
<Expr2> → <Expr1> <Expr2P>
<Expr2P> → <
Expr1> <Expr2P> | + <Expr1> <Expr2P> |λ
<Expr1> → <ExprUnaria> <Expr1P>
<Expr1P> → * <ExprUnaria> <Expr1P> | / <ExprUnaria> <Expr1P> | % <ExprUnaria> <Expr1P> |λ
<ExprUnaria> → + <ExprUnaria> | <
ExprUnaria> | ! <ExprUnaria> | <Primario>
<Primario> → **this**
<Primario> → <Literal>
<Primario> → (<Expresion>) <LlamadaL>

<Primario> → **new identificador** <ArgsActuales> <LlamadaL>
<Primario> → i**dentificador** <PrimarioP>
<PrimarioP> → <LlamadaL> | <ArgsActuales> <LlamadaL>
<LlamadaL> → <Llamada> <LlamadaL> |λ
<Llamada> → **. identificador** <ArgsOpcionales>
<ArgsOpcionales> → <ArgsActuales>
<ArgsOpcionales> → λ
<Literal> → **null** | **true** | **false** | **intLiteral** | **charLiteral** | **stringLiteral**
<ArgsActuales> → (<ArgsActualesP>
<ArgsActualesP> → <ListaExps>) |)
<ListaExps> → <Expresion> <ListaExpsP>
<ListaExpsP> → , <ListaExps> |λ

Ambigüedad de la gramática

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

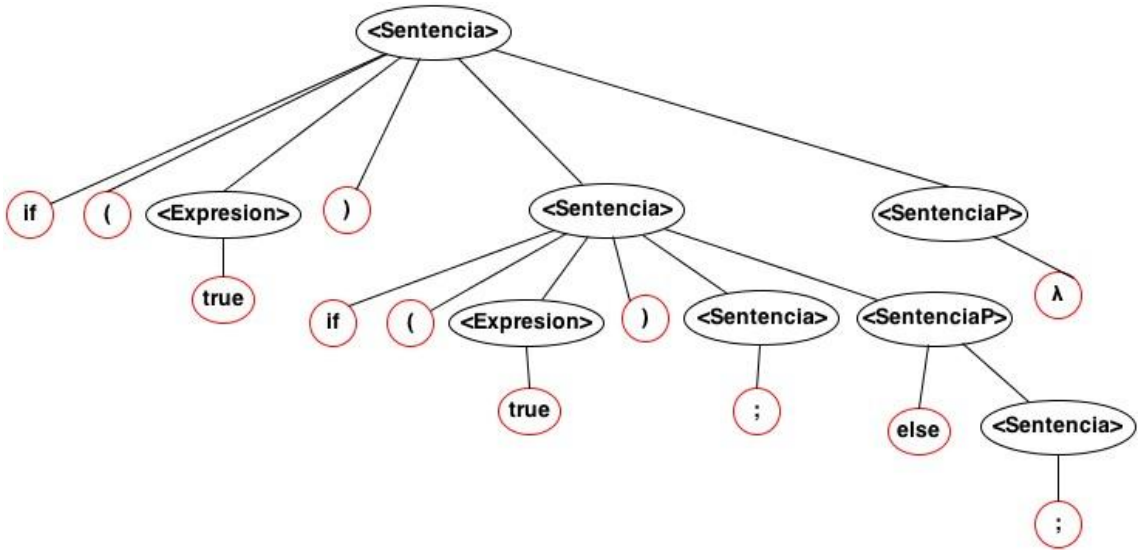
<Sentencia> → **if (** <Expresion> **)** <Sentencia> <SentenciaP>

<SentenciaP> → **else** <Sentencia> | λ

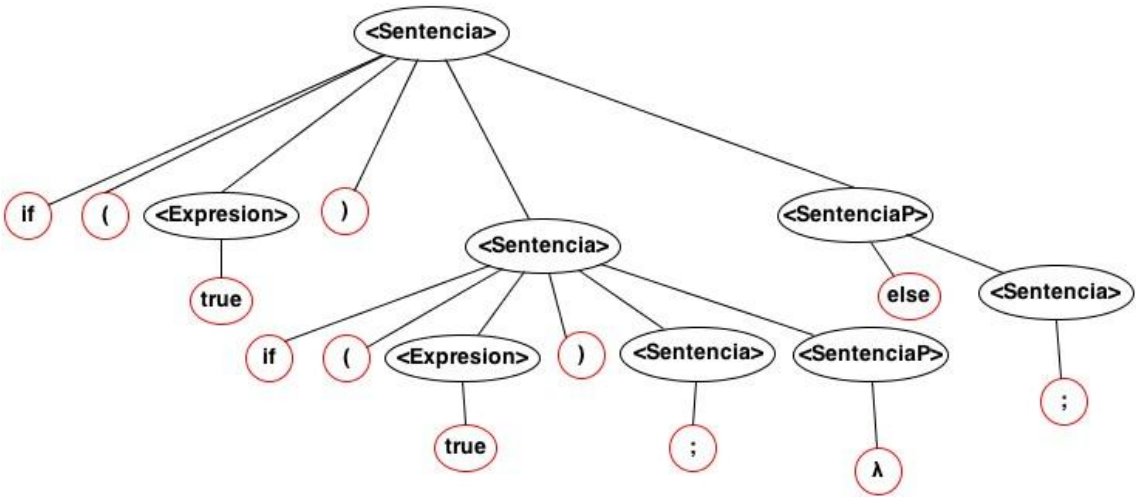
Justificación:

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 de 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.

Errores que se detectan.

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.

Analizador Semántico

Esquema de Traducción.

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> <InicialP>
<InicialP> → <Inicial>
<InicialP> → λ

<Clase> → **class** identificador {
```
                    <ClaseP>.class = TS.addClass(identificador.lexem)
```
} <ClaseP> {
```
        If (<ClaseP>.class.getConstructor() == null)
                    <ClaseP>.class.setConstructor(<ClaseP>.class.getClassID())
```
}

<ClaseP> → <Herencia> { `<ClaseP>.class.setAncestro(<Herencia>.ancestro)` } **{** {`<MiembroL>.class = <ClaseP>.class`} <MiembroL>**}**

<ClaseP> → {`<ClaseP>.class.setAncestro("Object")`} **{** {`<MiembroL>.class = <ClaseP>.class`} <MiembroL> **}**

<MiembroL> → {`<Miembro>.class = <MiembroL>.class`} <Miembro> {`<MiembroL>`$_1$`.class = <MiembroL>.class`} <MiembroL>$_1$

<MiembroL> → λ

<Herencia> → **extends** identificador {`<Herencia>.ancestro = identificador.lexem`}

<Miembro> → {`<Atributo>.class = <Miembro>.class`
```
                <Atributo>.metodo = null
```
} <Atributo>

<Miembro> → {`<Ctor>.class = <Miembro>.class`} <Ctor>

<Miembro> → {`<Metodo>.class = <Miembro>.class`} <Metodo>

<Atributo> → **varInst** <Tipo> {
```
                                        <ListaDecVars>.tipo = <Tipo>.tipo
```

<ListaDecVars>.class = <Atributo>.class
<ListaDecVars>.metodo = <Atributo>.metodo
<ListaDecVars>.es_varinst=<Atributo>.es_varinst
} <ListaDecVars> **;**

<Metodo> → <ModMetodo> <TipoMetodo> identificador {
<ArgsFormales>.metodo = <Metodo>.class.addMetodo(identificador.lexem, <ModMetodo>.mod, <TipoMetodo>.tipo)
} <ArgsFormales> {<Bloque>.metodo=<ArgsFormales>.metodo,<Bloque>.class=<Metodo>.class} <Bloque>

<Ctor> → identificador {
<ArgsFormales>.metodo = <Ctor>.class.setConstructor(identificador.lexem)
} <ArgsFormales> {<Bloque>.metodo=<ArgsFormales>.metodo,<Bloque>.class=<Ctor>.class} <Bloque>

<ArgsFormales> → **(** {<ArgsFormalesP>.metodo = <ArgsFormales>.metodo}<ArgsFormalesP>

<ArgsFormalesP> → {
<ListaArgsFormales>.metodo = <ArgsFormales>.metodo
<ListaArgsFormales>.indice = 0
} <ListaArgsFormales>)

<ArgsFormalesP> → **)**

<ListaArgsFormales> → <ArgFormal> {
<ListaArgsFormales>.metodo.addArgFormal(<ArgFormal>.id, <ArgFormal>.tipo, <ListaArgsFormales>.indice)
<ListaArgsFormalesP>.metodo = <ListaArgsFormales>.metodo
<ListaArgsFormalesP>.indice = <ListaArgsFormales>.indice+1
} <ListaArgsFormalesP>

<ListaArgsFormalesP> → **,** {<ListaArgsFormales>.metodo = <ListaArgsFormalesP>.metodo
<ListaArgsFormales>.indice = <ListaArgsFormalesP>.indice
} <ListaArgsFormales>

<ListaArgsFormalesP> → λ

<ArgFormal> → <Tipo> identificador {
<ArgFormal>.tipo = <Tipo>.tipo
<ArgFormal>.id = identificador.lexem }

<ModMetodo> → **static** {<ModMetodo>.mod = "static"}

<ModMetodo> → **dynamic** {<ModMetodo>.mod = "dynamic" }

<TipoMetodo> → <Tipo> {<TipoMetodo>.tipo = <Tipo>.tipo }

<TipoMetodo> → **void** {<TipoMetodo>.tipo = new TipoVoid() }
<Tipo> → <TipoPrimitivo > {<Tipo>.tipo = <TipoPrimitivo>.tipo}
<Tipo> → identificador {<Tipo>.tipo = new TipoClase(identificador.lex)}
<TipoPrimitivo> → **boolean** {<TipoPrimitivo>.tipo = new TipoBool()}
<TipoPrimitivo> → **char** {<TipoPrimitivo>.tipo = new TipoChar()}
<TipoPrimitivo> → **int** {<TipoPrimitivo>.tipo = new TipoInt()}
<TipoPrimitivo> → **String** {<TipoPrimitivo>.tipo = new TipoString()}
<ListaDecVars> → identificador {

```
if (<ListaDecVars>.class!=null)
      <ListaDecVars>.class.addAtributoInst(identificador.lex, <ListaDecVars>.tipo)
else
      <ListaDecVars>.metodo.addVarLocal(identificador.lex, <ListaDecVars>.tipo)

<ListaDecVarsP>.class = <ListaDecVars>.class
<ListaDecVarsP>.metodo = <ListaDecVars>.metodo
<ListaDecVarsP>.tipo = <ListaDecVars>.tipo
<ListaDecVarsP>.es_varinst= <ListaDecVars>.es_varinst
```

} <ListaDecVarsP>

<ListaDecVarsP> → **,** {

```
<ListaDecVars>.class = <ListaDecVarsP>.class
<ListaDecVars>.metodo = <ListaDecVarsP>.metodo
<ListaDecVars>.tipo = <ListaDecVarsP>.tipo
<ListaDecVars>.es_varinst = <ListaDecVarsP>.es_varinst
```

} <ListaDecVars>

<ListaDecVarsP> → λ
<Bloque> → **{** {<SentenciaL>.bloque = new Bloque(),<SentenciaL>.class=<Bloque>.class,<SentenciaL>.metodo=<Bloque>.metodo}
<SentenciaL> **}** {<Bloque>.bloque = <SentenciaL>.bloque}
<SentenciaL> → {<Sentencia>.metodo=<SentenciaL>.metodo, <Sentencia>.class=<SentenciaL>.class} <Sentencia> {
<SentenciaL>.bloque.addSent(<Sentencia>.sent)} $<\text{SentenciaL}>_1$
<SentenciaL> → λ
<Sentencia> → **;**
<Sentencia> → <Asignacion> {<Sentencia>.sent = <Asignacion>.sent}

<Sentencia> → <SentenciaSimple> {<Sentencia>.sent=<SentenciaSimple>.sent} ;

<Sentencia> → varLocal <Tipo> {ListaDecVars>.tipo=<Tipo>.tipo , <ListaDecVars>.metodo=<Sentencia>.metodo, <ListaDecVars>.class=<Sentencia>.class}<ListaDecVars>;

<Sentencia> → **if (**<Expresion>**)**{<Sentencia1>.class=<Sentencia>.class, <Sentencia1>.metodo=<Sentencia>.metodo} <Sentencia>$_1$ {<SentenciaP>.class=<Sentencia>.class, <SentenciaP>.metodo=<Sentencia>.metodo, <SentenciaP>.class=<Sentencia>.class}<SentenciaP> {

```
if(<SentenciaP>.sent!=null)
     <Sentencia>.sent = new NodoIf(<Expresion>.expr, <Sentencia>1.sent, <SentenciaP>.sent)
else
      <Sentencia>.sent = new NodoIf(<Expresion>.expr, <Sentencia>1.sent)}
```

<SentenciaP> → **else** {<Sentencia>.class=<SentenciaP>.class, <Sentencia>.metodo=<SentenciaP>.metodo} <Sentencia> {<SentenciaP>.sent = <Sentencia>.sent}

<SentenciaP> → { <SentenciaP>.sent = null }

<Sentencia> → **while (**<Expresion>**)** {<Sentencia1>.class=<Sentencia>.class, <Sentencia1>.metodo=<Sentencia>.metodo} <Sentencia>$_1$ {<Sentencia>.sent = new NodoWhile(<Expresion>.expr, <Sentencia>1.sent)}

<Sentencia> → **for (**<Asignacion> $_1$**;** <Expresion> **;** <Asignacion>$_2$**)** {<Sentencia1>.class=<Sentencia>.class, <Sentencia1>.metodo=<Sentencia>.metodo}<Sentencia>$_1$

```
     {<Sentencia>.sent = new NodoFor
(<Asignacion>1.sent,<Expresion>.expr,<Asignacion>2.sent,<Sentencia>1.sent)}
```

<Sentencia> →{<Bloque>.class=<Sentencia>.class, <Bloque>.metodo=<Sentencia>.metodo} <Bloque> {<Sentencia>.sent = new NodoBloque(<Bloque>.bloque)}

<Sentencia> → **return** <SentenciaPP> {<Sentencia>.sent= SentenciaPP>.sent}

<SentenciaPP> → <Expresion> {<SentenciaPP>.sent = new NodoReturn(<Expresion>.expr) };

<SentenciaPP**>** → { <SentenciaPP>.sent = new NodoReturn()} ;

<Asignacion> → <LadoIzquierdo> = <Expresion> {<Asignacion>.sent = new NodoAsignacion(<LadoIzquierdo>.lista_ids,<Expresion>.expr)}

<LadoIzquierdo> → identificador {<LadoIzquierdo>.lista_ids.setEncadenado(identificador.lexema)} <IdEncadenados> {<LadoIzquierdo>.lista_ids = <IdEncadenados>.lista_ids}

<LadoIzquierdo> → identificador . {<LadoIzquierdo>.lista_ids.setEncadenado(identificador.lexema), <LadoIzquierdo>$_1$.lista_ids=<LadoIzquierdo>.lista_ids} <LadoIzquierdo>$_1$

<IdEncadenados> → λ

<IdEncadenados> → . Identificador {<IdEncadenados>.lista_ids.agregar(identificador.lexema)} <IdEncadenados>$_1$ {<IdEncadenados>.lista_ids=<IdEncadenados>$_1$.lista_ids}

<SentenciaSimple> → **(**<Expresion> {<SentenciaSimple>.sent = <Expresion>.expr} **)**

<Expresion> → <Expr5> {<ExprP>.exprH = <Expr5>.expr } <ExprP> {<Expresion>.expr = <ExprP>.expr}

<ExprP> → **||** <Expr5> {<ExprP>1.exprH = new NodoExpBinaria(<ExprP>.exprH, <Expr5>.expr, "||")} <ExprP>$_1$ {<ExprP>.expr = <ExprP>1.expr}

<ExprP> → λ {<ExprP>.expr = <ExprP>.exprH}

<Expr5> → <Expr4>{<Expr5P>.exprH = <Expr4>.expr } <Expr5P> {<Expr5>.expr = <Expr5P>.expr}

<Expr5P> → **&&** <Expr4>{<Expr5P>1.exprH = new NodoExpBinaria(<Expr5P>.exprH, <Expr4>.expr, "&&")}<Expr5P>$_1$ {<Expr5P>.expr = <Expr5P>1.expr}

<Expr5P> → λ {<Expr5P>.expr = <Expr5P>.exprH}

<Expr4> → <Expr3> {<Expr4P>.exprH = <Expr3>.expr } <Expr4P> {<Expr4>.expr = <Expr4P>.expr}

<Expr4P> → **==** <Expr3> {<Expr4P>1.exprH = new NodoExpBinaria(<Expr4P>.exprH, <Expr3>.expr, "==")}<Expr4P>$_1$ {<Expr4P>.expr = <Expr4P>1.expr}

<Expr4P> → **!=** <Expr3> {<Expr4P>1.exprH = new NodoExpBinaria(<Expr4P>.exprH, <Expr3>.expr, "!=")}<Expr4P>$_1$ {<Expr4P>.expr = <Expr4P>1.expr}

<Expr4P> → λ {<Expr4P>.expr = <Expr4P>.exprH}

<Expr3> → <Expr2> {<Expr3P>.exprH = <Expr2>.expr } <Expr3P> {<Expr3>.expr = <Expr3P>.expr}

<Expr3P> → **>=** <Expr2> {<Expr3P>.expr = new NodoExpBinaria(<Expr3P>.exprH, <Expr2>.expr, ">=")}

<Expr3P> → **<=** <Expr2> {<Expr3P>.expr = new NodoExpBinaria(<Expr3P>.exprH, <Expr2>.expr, "<=")}

<Expr3P> → **>** <Expr2> {<Expr3P>.expr = new NodoExpBinaria(<Expr3P>.exprH, <Expr2>.expr, ">")}

<Expr3P> → **<** <Expr2> {<Expr3P>.expr = new NodoExpBinaria(<Expr3P>.exprH, <Expr2>.expr, "<")}

<Expr3P> → λ {<Expr3P>.expr = <Expr3P>.exprH}

<Expr2> → <Expr1>{<Expr2P>.exprH = <Expr1>.expr } <Expr2P> {<Expr2>.expr = <Expr2P>.expr}

<Expr2P> → **-** <Expr1> {<Expr2P>1.exprH = new NodoExpBinaria(<Expr2P>.exprH, <Expr1>.expr, "-")}<Expr2P>$_1$ {<Expr2P>.expr = <Expr2P>1.expr}

<Expr2P> → **+** <Expr1>{<Expr2P>1.exprH = new NodoExpBinaria(<Expr2P>.exprH, <Expr1>.expr, "+")}<Expr2P>$_1$ {<Expr2P>.expr = <Expr2P>1.expr}

<Expr2P> → λ {<Expr2P>.expr = <Expr2P>.exprH}

<Expr1> → <ExprUnaria>{<Expr1P>.exprH = <ExprUnaria>.expr } <Expr1P> {<Expr1>.expr = <Expr1P>.expr}

<Expr1P> → * <ExprUnaria>{<Expr1P>1.exprH = new NodoExpBinaria(<Expr1P>.exprH, <ExprUnaria>.expr, "*")}<Expr1P>$_1$ {<Expr1P>.expr = <Expr1P>1.expr}

<Expr1P> → / <ExprUnaria>{<Expr1P>1.exprH = new NodoExpBinaria(<Expr1P>.exprH, <ExprUnaria>.expr, "/")}<Expr1P>$_1${<Expr1P>.expr = <Expr1P>1.expr}

<Expr1P> → % <ExprUnaria> {<Expr1P>1.exprH = new NodoExpBinaria(<Expr1P>.exprH, <ExprUnaria>.expr, "%")}<Expr1P>$_1$ {<Expr1P>.expr = <Expr1P>1.expr}

<Expr1P> → {<Expr1P>.expr = <Expr1P>.exprH}

<ExprUnaria> → + <ExprUnaria>$_1$ {<ExprUnaria>.expr = new NodoExpUnaria(<ExprUnaria>1.expr, "+")}

<ExprUnaria> → - <ExprUnaria>$_1$ {<ExprUnaria>.expr = new NodoExpUnaria(<ExprUnaria>1.expr, "-")}

<ExprUnaria> → ! <ExprUnaria>$_1$ {<ExprUnaria>.expr = new NodoExpUnaria(<ExprUnaria>1.expr, "!")}

<ExprUnaria> → <Primario> {<ExprUnaria>.expr = <Primario>.prim}

<Primario> → **this** {<Primario>.prim = new NodoPrimThis()}

<Primario> → <Literal> {<Primario>.prim = <Literal>.prim}

<Primario> → **(** <Expresion> **)** <LlamadaL> {<Primario>.prim= new NodoPrimParentizado(<Expresion>.expr, <LlamadaL>.llamadas)}

<Primario> → **new** identificador <ArgsActuales> <LlamadaL> {<Primario>.prim = new NodoPrimNew(identificador.lexem, <ArgsActuales>.args, <LlamadaL>.llamadas)}

<Primario> → identificador {<PrimarioP>.id = identificador.lexem} <PrimarioP> {<Primario>.prim = <PrimarioP>.prim}

<PrimarioP> → <LlamadaL> {<PrimarioP>.prim = new NodoIdDirecto(<PrimarioP>.id, <LlamadaL>.llamadas)}

<PrimarioP> → <ArgsActuales> <LlamadaL> {<PrimarioP>.prim = new NodoLlamadaDirecta(<PrimarioP>.id, <ArgsActuales>.args,<LlamadaL>.llamadas)}

<LlamadaL> → <Llamada> {
```
<LlamadaL>.llamadas.setEncadenado(<Llamada>.encadenado)
<LlamadaL>1.llamadas = <LlamadaL>.llamadas
```
} <LlamadaL>$_1$

<LlamadaL> → λ

<Llamada> → .identificador <ArgsOpcionales> {if (<argsOpcionales>.args ==null)
<Llamada>.encadenado = new NodoIdEncadenadoDer(identificador.lexem)}
else
<Llamada>.encadenado = new LlamadaEncadenada(identificador.lexem, <ArgsOpcionales>.args)

<ArgsOpcionales> → {<ArgsOpcionales>.args = <ArgsActuales>.args)}<ArgsActuales>
< ArgsOpcionales > → λ
<Literal> → **null** {<Literal>.prim = new NodoPrimLiteral(null.lexem, new TipoClase("Object"))}
<Literal> → **true** {<Literal>.prim = new NodoPrimLiteral(true.lexem, new TipoBool())}
<Literal> → **false** {<Literal>.prim = new NodoPrimLiteral(false.lexem, new TipoBool())}
<Literal> → **intLiteral** {<Literal>.prim = new NodoPrimLiteral(intLiteral.lexem, new TipoInt())}
<Literal> → **charLiteral** {<Literal>.prim = new NodoPrimLiteral(charLiteral.lexem, new TipoChar())}
<Literal> → **stringLiteral** {<Literal>.prim = new NodoPrimLiteral(stringLiteral.lexem, new TipoString())}
<ArgsActuales> → **(** <ArgsActualesP> {<ArgsActuales>.args = <ArgsActualesP>.args}
<ArgsActualesP> → {<ListaExps>.args = new ListaArgs()} <ListaExps> {<ArgsActualesP>.args = <ListaExps>.args} **)**
<ArgsActualesP> → **)** {<ArgsActualesP>.args = new ListaArgs()}
<ListaExps> → <Expresion> {
<ListaExps>.args.addArg(<Expresion>.expr)
<ListaExpsP>.args = <ListaExps>.args
} <ListaExpsP>
<ListaExpsP> → **,** {<ListaExps>.args = <ListaExpsP>.args} <ListaExps>
<ListaExpsP> → λ

Atributos utilizados en el EdT.

Atributo: class

Utilizado en:<ClaseP>, <Miembro>, <MiembroL>, <Atributo>, <Ctor>, <Metodo>, <ListaDecVars>, <ListaDecVarsP> <Bloque>, <SentenciaL> , <Sentencia> , <SentenciaP>

Tipo: Clase

Descripción: atributo heredado utilizado para almacenar una clase.

Atributo: metodo

Utilizado en: <Atributo>, <ArgsFormales>, <VarsLocales>, <ListaArgsFormales>, <ListaArgsFormalesP>, <ListaDecVars>, <ListaDecVarsP>, <Bloque>,<SentenciaL> , <Sentencia> , <SentenciaP>

Tipo: Metodo

Descripción: atributo heredado utilizado para mantener la referencia a un método.

Atributo: tipo

Utilizado en: <ListaDecVars>, <Tipo>, <TipoMetodo>, <ArgFormal>, <TipoPrimitivo>, <ListaDecVarsP>

Tipo: Tipo

Descripción: Es un atributo heredado para <ListaDecVars> y <ListaDecVarsP> utilizado para almacenar el tipo, y es un atributo sintetizado para el resto y es utilizado para la misma función.

Atributo: índice

Utilizado en: <ListaArgsFormales>, <ListaArgsFormalesP>

Tipo: entero

Descripción: atributo heredado utilizado para almacenar la posición en la cual son declarados los argumentos.

Atributo: id

Utilizado en: <ArgFormal>, <PrimarioP>,

Tipo: String

Descripción: atributo heredado para <PrimarioP> y sintetizado para <ArgFormal> y es utilizado para almacenar el lexema de un Token identificador.

Atributo: mod

Utilizado en: <ModMetodo>

Tipo: String

Descripción: atributo sintetizado utilizado para almacenar el modificador de los métodos.

Atributo: bloque

Utilizado en: <SentenciaL>, <Bloque>

Tipo: Bloque

Descripción: atributo heredado para <SentenciaL> y sintetizado para <Bloque>. Es utilizado para almacenar un objeto de tipo bloque.

Atributo: sent
Utilizado en: <Sentencia>, <SentenciaSimple>, <Asignacion>, <SentenciaP>, <SentenciaPP>
Tipo: NodoSentencia
Descripción: atributo sintetizado que es utilizado para almacenar un objeto de tipo NodoSentencia, usado en la construcción del AST.

Atributo: expr
Utilizado en: <Expresion>, <Expr5>, <ExprP>, <Expr4>, <Expr5P>, <Expr3>, <Expr4P>, <Expr2>, <Expr3P>, <Expr2P>, <Expr1>, <Expr1P>, <ExprUnaria>
Tipo: NodoExpresion
Descripción: atributo sintetizado que almacena una expresión.

Atributo: exprH
Utlizado en: <ExprP>, <Expr5P>, <Expr4P>, <Expr3P>, <Expr2P>, <Expr1P>
Tipo: NodoExpresion
Descripción: atributo heredado utilizado para la construcción de expresiones.

Atributo: prim
Utlizado en: <Primario>, <PrimarioP>, <Literal>
Tipo: NodoExpPrimario
Descripción: atributo sintetizado utilizado para almacenar un nodo primario.

Atributo: args
Utilizado en: <ArgsActuales>, <ArgsActualesP>, <ListaExps>, <ListaExpsP>
Tipo: ListaArgs
Descripción: atributo heredado que sirve para almacenar una lista de argumentos actuales.

Atributo: es_varinst
Utilizado en: <ListaDecVars>, <ListaDecVarsP> <Atributo>
Tipo: boolean
Descripcion: atributo heredado utilizado para saber si es una variable de instancia o local.

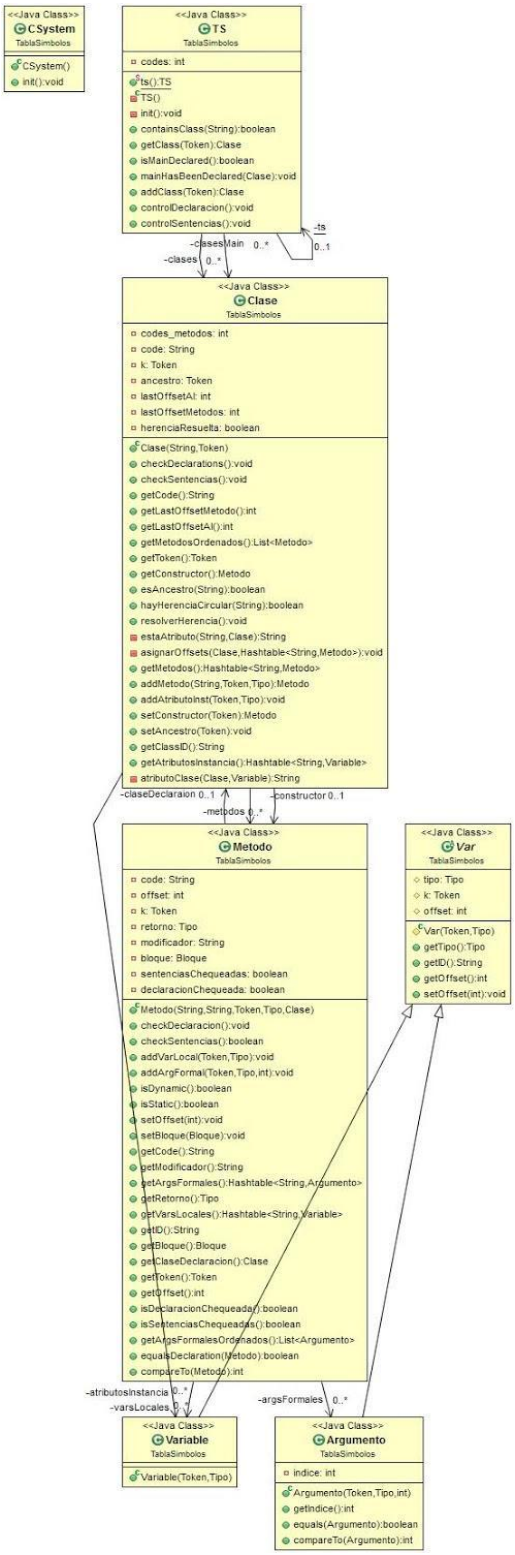
Atributo: lista_ids
Utilizado en: <LadoIzquierdo> , <IdEncadenados>
Tipo: NodoId
Descripcion: atributo sintetizado utilizado para almacenar identificadores encadenados.

Atributo: encadenado
Utilizado en: <Llamada>
Tipo: Encadenado
Descripcion: atributo sintetizado utilizado para almacenar una llamada

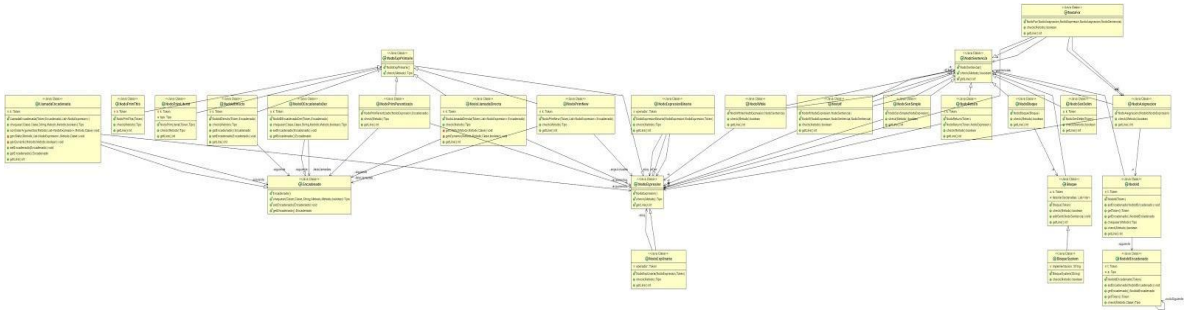
Diagrama de Clases

Se presentará en primer lugar el diagrama de clases correspondiente a la tabla de símbolos y luego los diagramas correspondientes al AST finalizando con aquellos diagramas con clases comunes a tanto al AST como a la TS.

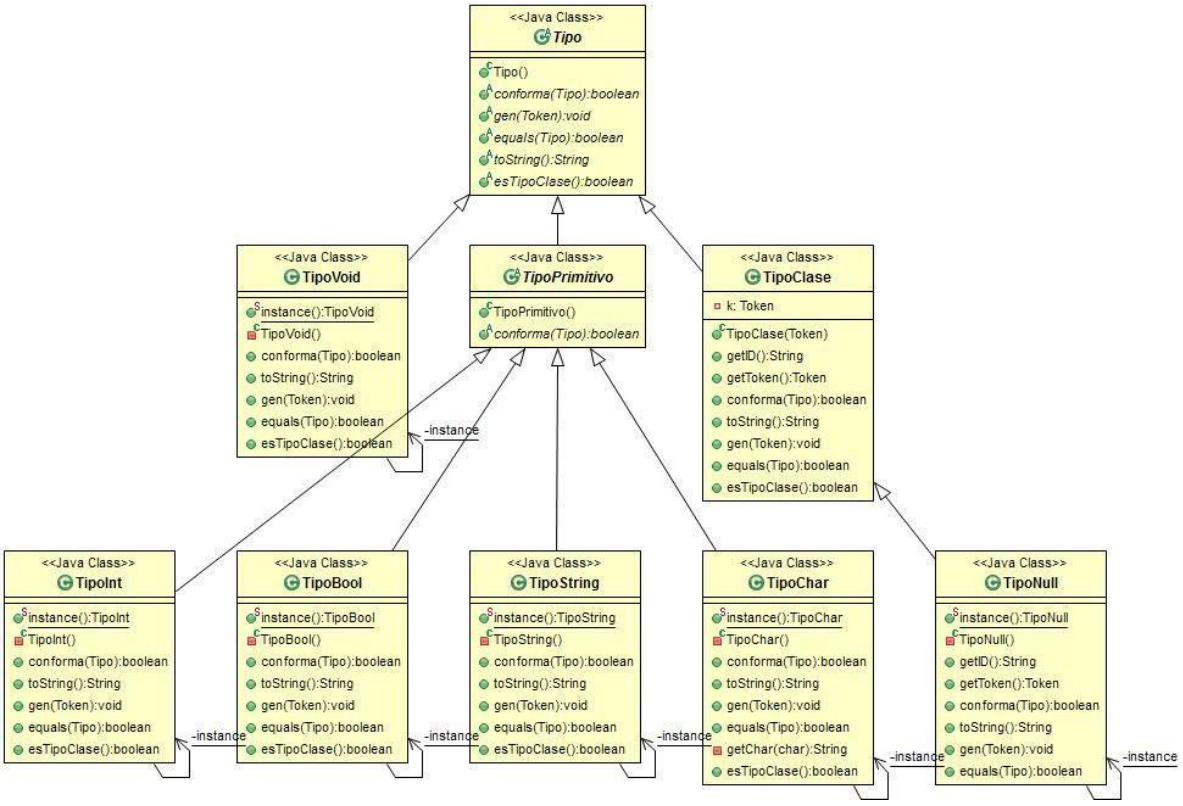
Diagrama de clases TS


<<Java Class>>
CSystem
TablaSimbolos
CSystem()
init():void
<<Java Class>>
TS
TablaSimbolos
codes: int
ts():TS
TS()
init():void
containsClass(String):boolean
getClass(Token):Clase
isMainDeclared():boolean
mainHasBeenDeclared(Clase):void
addClass(Token):Clase
controlDeclaracion():void
controlSentencias():void
-ts
0..1
-clasesMain 0..*
-clases 0..*
<<Java Class>>
Clase
TablaSimbolos
codes_metodos: int
code: String
k: Token
ancestro: Token
lastOffsetAI: int
lastOffsetMetodos: int
herenciaResuelta: boolean
Clase(String,Token)
checkDeclarations():void
checkSentencias():void
getCode():String
getLastOffsetMetodo():int
getLastOffsetAI():int
getMetodosOrdenados():List<Metodo>
getToken():Token
getConstructor():Metodo
esAncestro(String):boolean
hayHerenciaCircular(String):boolean
resolverHerencia():void
estaAtributo(String,Clase):String
asignarOffsets(Clase,Hashtable<String,Metodo>):void
getMetodos():Hashtable<String,Metodo>
addMetodo(String,Token,Tipo):Metodo
addAtributoInst(Token,Tipo):void
setConstructor(Token):Metodo
setAncestro(Token):void
getClassID():String
getAtributosInstancia():Hashtable<String,Variable>
atributoClase(Clase,Variable):String
-claseDeclaraion 0..1
-metodos 0..*
-constructor 0..1
<<Java Class>>
Metodo
TablaSimbolos
code: String
offset: int
k: Token
retorno: Tipo
modificador: String
bloque: Bloque
sentenciasChequeadas: boolean
declaracionChequeada: boolean
Metodo(String,String,Token,Tipo,Clase)
checkDeclaracion():void
checkSentencias():boolean
addVarLocal(Token,Tipo):void
addArgFormal(Token,Tipo,int):void
isDynamic():boolean
isStatic():boolean
setOffset(int):void
setBloque(Bloque):void
getCode():String
getModificador():String
getArgsFormales():Hashtable<String,Argumento>
getRetorno():Tipo
getVarsLocales():Hashtable<String,Variable>
getID():String
getBloque():Bloque
getClaseDeclaracion():Clase
getToken():Token
getOffset():int
isDeclaracionChequeada():boolean
isSentenciasChequeadas():boolean
getArgsFormalesOrdenados():List<Argumento>
equalsDeclaration(Metodo):boolean
compareTo(Metodo):int
<<Java Class>>
Var
TablaSimbolos
tipo: Tipo
k: Token
offset: int
Var(Token,Tipo)
getTipo():Tipo
getID():String
getOffset():int
setOffset(int):void
-atributosInstancia 0..*
-varsLocales 0..*
-argsFormales 0..*
<<Java Class>>
Variable
TablaSimbolos
Variable(Token,Tipo)
<<Java Class>>
Argumento
TablaSimbolos
indice: int
Argumento(Token,Tipo,int)
getIndice():int
equals(Argumento):boolean
compareTo(Argumento):int

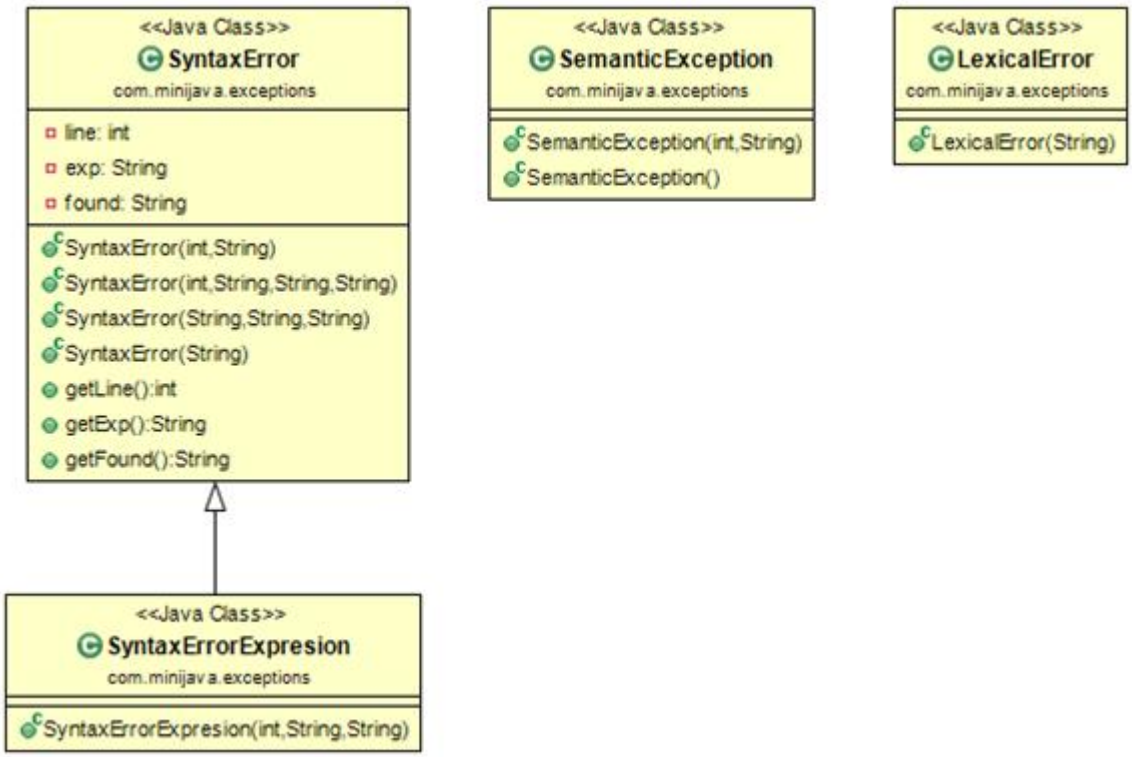
Diagramas de clases AST.



Diagramas de Tipos.



Diagramas de clases comunes a la TS y al AST.



Control de Sentencias y Declaraciones

Para el control de sentencias y declaraciones se utilizaron los siguientes métodos:

Clase.class

public void checkDeclarations() throws SemanticException {…}
Donde allí:

- Si es la clase Object, esta correcta.
- Chequeo las declaraciones de variables de instancia
- Controlo que no halla herencia circular y ademas controla que las clases implicadas en la herencia esten declaradas.
- Resuelvo la herencia de los métodos
- Realizo el chequeo de declaracion de los métodos (llamo al Check de los métodos)
- Si esta clase tiene un metodo main() entonces le indica al TS que esta clase tiene un metodo main.

public void checkSentencias() throws SemanticException {…}
Donde allí:

- Chequeo la correctitud del constructor
- Chequeo el cuerpo de los metodos (llamo al check de los métodos)

Metodo.class

public void checkDeclaracion() throws SemanticException {…}
Donde allí:

- Si el retorno es de tipo clase, controlo que dicha clase debe estar declarada
- Verifica que para los argumentos de tipo clase, la misma esté declarada.
- Verifica que para las variables locales de tipo clase, la misma este declarada.

public boolean checkSentencias() throws SemanticException {…}

- Chequea las sentencias correspondientes al bloque de este método

Errores semánticos que se detectan

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.**
- No pueden declararse **métodos con el mismo nombre que la clase.**
- No pueden usarse nombres de variables de instancia o referencias a *this* en las expresiones del cuerpo de métodos.
- No puede haber **mas de un constructor por clase.**
- Un método/constructor **no puede tener mas 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.**
- Los tipos de las expresiones deben **conformar.**
- Las sentencias deben ser **correctamente tipadas.**
- Detecta **código inalcanzable.**
- No puede llamarse un método dinámico en un contexto estático

Todos estos errors fueron vinculados a la clase “SemanticException” quien se encarga de arrojar el mensaje de error apropiado para cada tipo de error.

Desiciones de diseño.

· Se admite que mas de una clase defina un método “main”, ejecutándose el primero encontrado. Sin embargo, se arrojará por pantalla una advertencia informando que se ha declarado mas de un método “main” y qué clases lo hicieron.

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.

Asignación de offsets.

Metodo.class

Asignacion de offsets para variables locales: Los offsets son asignados a medida que van apareciendo en el análisis sintáctico. Se asignan valores consecutivos partiendo desde cero.

```
void addVarLocal(Token k, Tipo tipo) throws SemanticException {
…
        v.setOffset(-(varsLocales.size() - 1));
…
}
```

Asignación de offsets para argumentos formales: Los offsets son asignados al momento de realizar el chequeo de sentencias, se realiza en este instante debido a que es necesario conocer la cantidad de argumentos formales de los métodos para agregarlos en la pila, en el orden correspondiente (del último declarado al primero). Si el método es dinámico se asignan valores consecutivos a partir de 4, de otra forma, si el método es estático se asignan valores consecutivos a partir de 3.
Esto es debido a que en un método estático el RA (registro de activación) se compone del enlace dinámico (ED), seguido del puntero de retorno (PR), y luego los argumentos. En cambio, en un método dinámico se almacena un lugar extra para el objeto actual THIS, considerando también el RA, ED y PR.

```
public boolean checkSentencias() throws SemanticException {
…
        for (int i = 0; i < argsFormalesList.size(); i++) {
      if (this.isDynamic())
        argsFormalesList.get(i).setOffset(argsFormalesList.size() + 3-i);
      else
        argsFormalesList.get(i).setOffset(argsFormalesList.size() + 2-i);
    }
…
}
```

Aclaración: para hacer hincapié solo en las instrucciones correspondientes a la asignación de offsets, se eliminaron aquellas que no agregan nada a dicha asignación.

Clase.class

Asignación de offsets para métodos y variables de instancias: Se le asignan los offsets a los métodos en la clase en la que son declarados y dicho offset se mantiene a través de la línea de herencia, correspondiente a dicha clase. Esto se logra con el uso de la variable **lastOffsetMetodos** (el cual almacena el último offset asignado, esto es útil para que una clase hija conozca el valor a partir del cual deberá asignar el offset a los métodos en ella declarada)**.** Lo mismo sucede con las variables de instancia, en este caso la variable involucrada es **lastOffsetAI.**
La asignación de offset de los métodos es a partir del valor 0, y el de las variables de instancia a partir del valor 1 (debido al puntero a la VT en el CIR).

```
private void asignarOffsets(Clase superClase, Hashtable<String, Metodo> metodosAncestro) {
		this.lastOffsetAI = superClase.getLastOffsetAI();
		this.lastOffsetMetodos = superClase.getLastOffsetMetodo();
		int offsetStatics = 0;
		Metodo mAncestro;
		for (Metodo m : this.getMetodos().values()) {
			if (m.isDynamic()) {
				if (!metodosAncestro.containsKey(m.getID()))
					m.setOffset(lastOffsetMetodos++);
				else {
					mAncestro = metodosAncestro.get(m.getID());
					m.setOffset(mAncestro.getOffset());
				}
			} else
				m.setOffset(offsetStatics++);
		}

		for (Variable v : this.getAtributosInstancia().values())
			v.setOffset(this.lastOffsetAI++);

	}
```

Con el control de sentencias se inicia la generación de código propiamente dicha.

TS.class

```
/* Protocolo de inicializacion */
 Date d = new Date();
 GCI.gen().comment("# Codigo genenerado por el compilador minijava");
 GCI.gen().comment("# Generado: " + d.toString());
 GCI.gen().comment("# Autores:\tFrancisco Cuenca");
 GCI.gen().comment("# \t\t\tBrenda Dilschneider");
 GCI.gen().comment("# Compiladores e Interpretes 2014");
 GCI.gen().comment("# DCIC UNS - Argentina");
 GCI.gen().ln();
```

```
 GCI.gen().ln();
 GCI.gen().comment("<<<<< Codigo de inicializacion de la maquina virtual >>>>");
 GCI.gen().ln();
 GCI.gen().code();
 GCI.gen().gen("PUSH lheap_init","");
 GCI.gen().gen("CALL","");
 GCI.gen().gen("PUSH "+clasesMain.get(0).getMetodos().get("main").getCode(),"");
 GCI.gen().gen("CALL","");
 GCI.gen().gen("HALT","");
 GCI.gen().ln();
 GCI.gen().gen("lmalloc", "LOADFP", "Inicializacion unidad");
 GCI.gen().gen("LOADSP","");
 GCI.gen().gen("STOREFP","Finaliza inicializacion del RA");
 GCI.gen().gen("LOADHL","hl");
 GCI.gen().gen("DUP","hl");
 GCI.gen().gen("PUSH 1","1");
 GCI.gen().gen("ADD","hl + 1");
 GCI.gen().gen("STORE 4","Guarda resultado (puntero a base del bloque)");
 GCI.gen().gen("LOAD 3","Carga cantidad de celdas a alojar (parametro)");
 GCI.gen().gen("ADD","");
 GCI.gen().gen("STOREHL","Mueve el heap limit (hl)");
 GCI.gen().gen("STOREFP","");
 GCI.gen().gen("RET 1","Retorna eliminando el parametro");
 GCI.gen().ln();
 GCI.gen().gen("lheap_init","RET 0","Inicializacion simplicada del :heap");
 GCI.gen().ln();
 GCI.gen().ln();
 GCI.gen().comment("<<<< Inicio de generacion de codigo del progama fuente.
>>>>");
 for (Clase c : clases.values()) {
       c.checkSentencias();
 }

 System.out.println("El control de sentencias ha finalizado con exito.");
  }
 }
```

Clase.class

```
public void checkSentencias() throws SemanticException {
        boolean hayReturn = false;
        List<Metodo> metodosOrdenados = getMetodosOrdenados();

        GCI.gen().ln();
        GCI.gen().ln();
        GCI.gen().comment("<<<<  Clase " + getClassID() + " >>>>");
        GCI.gen().comment("Metodos dinamicos en la Virtual Table");
        String s = "DW ";
        for (Metodo m : metodosOrdenados)
                if (m.isDynamic()) {
                GCI.gen().comment("<" + m.getCode() + "> \tID: " + m.getID()
                + " \t> offset: " + m.getOffset());
                s += m.getCode() + ", ";
```

```
		}
	// Hay almenos un metodo dinamico.
	if (s.length() > 3) {
		GCI.gen().ln();
		GCI.gen().data();

		String ss = s.substring(0, s.length() - 2);
		GCI.gen().gen(getCode(), ss, "");
		GCI.gen().ln();
	} else{
		GCI.gen().ln();
		GCI.gen().data();
		GCI.gen().comment("No hay metodos dinamicos, creo la Virtual Table Vacia");
		GCI.gen().gen(getCode(), "NOP", "<VT Vacia>");
	}
	GCI.gen().ln();

	/* CONTROL DE SENTENCIAS */
	GCI.gen().code();
	GCI.gen().gen(constructor.getCode(), "NOP", "<Constructor>");
	hayReturn = constructor.checkSentencias();
	GCI.gen().ln();

	if (hayReturn)
		throw new SemanticException(k.getLine(),
			"Un constructor no puede contener una sentencia return.");

	// check del cuerpo de los metodos.
	for (Metodo m : metodosOrdenados) {
		if (!m.isSentenciasChequeadas()) {
		GCI.gen().gen(m.getCode(), "NOP", "<Metodo "+ m.getID()+">");
		hayReturn = m.checkSentencias();

	// si m es una funcion y no hay un return entonces es un error.
		if (!hayReturn && !(m.getRetorno() instanceof TipoVoid))
		throw new SemanticException(m.getToken().getLine(),
		"Falta sentencia return en el metodo " + m.getID()
		+ ".");
		GCI.gen().ln();
		GCI.gen().ln();
		}
	}
	}
```

Metodo.class

```
public boolean checkSentencias() throws SemanticException {
  this.sentenciasChequeadas = true;
```

```
/*
* Asignar offsets de los argumentos formales.
*
* El offset del i-esimo argumento se calcula como (n + k -i) donde n es
* la cantidad de argumentos formales del metodo. Si este metodo es
* dinamico k=3 (Enlace dinamico + Puntero de retorno + puntero a THIS)
* ; Si este metodo es estatico k=2 (Enlace dinamico + Puntero de
* retorno)
*/
        List<Argumento> argsFormalesList = getArgsFormalesOrdenados();
         for (int i = 0; i < argsFormalesList.size(); i++) {
                if (this.isDynamic())
                        argsFormalesList.get(i).setOffset(argsFormalesList.size() + 3 - i);
                 else
                        argsFormalesList.get(i).setOffset(argsFormalesList.size() + 2 - i);
        }

    GCI.gen().gen("LOADFP", "Guardar enalce dinamico");
    GCI.gen().gen("LOADSP", "Inicializar el FP");
    GCI.gen().gen("STOREFP", "");
   // Se reserva espacio para las variables locales.
    if (this.varsLocales.size() > 0)
          GCI.gen().gen("RMEM " + this.varsLocales.size(), "se reserva espacio para las variables
locales.");
// se hace un check del bloque
    boolean ret = getBloque().check(this);
// Se liberan el espacio reservado para las variables locales.
    if (this.getVarsLocales().size() > 0)
           GCI.gen().gen("FMEM " + this.getVarsLocales().size(),
          "Libera de la memoria las variables locales del metodo <" +
          this.getClaseDeclaracion().getClassID() + "::"+ this.getID() + ">");
// Se reestablece el contexto, y se transfiere el control para retomar
// la unidad invocadora.
  GCI.gen().gen("STOREFP", "Reestablece el contexto.");
// Retorno de la unidad, liberando el espacio utilizado por los
// argumentos formales.
// Si es dinamico es un +1 por el THIS.
  if (this.isDynamic())
```

```
        GCI.gen().gen("RET " + (this.getArgsFormales().size() + 1),"Retorna liberando de la
        memoria los argumentos, y el THIS del metodo <" +
        this.getClaseDeclaracion().getClassID()+ "::" + this.getID() + ">");
else
         GCI.gen().gen( "RET " + this.getArgsFormales().size(),
        "Retorna liberando de la memoria los argumentos del metodo <" +
        this.getClaseDeclaracion().getClassID() + "::"+ this.getID() + ">");
return ret;
}
```

TipoBool.class

```
@Override
 public void gen(Token k) {
     int t = k.getLexema().equals("true") ? 1 : 0;
     GCI.gen().gen("PUSH " + t, "Apila el literal <" + (t == 1 ? "True" : "False")+">");
  }
```

TipoChar.class

```
@Override
  public void gen(Token k) {
          GCI.gen().gen("PUSH " + ((int)k.getLexema().charAt(0)),"Apila el literal caracter
        <"+k.getLexema()+">");
  }
```

TipoClase.class

TipoVoid.class

```
@Override
public void gen(Token k) {
 }
```

TipoNull.class

```
@Override
   public void gen(Token k) {
       GCI.gen().gen("PUSH 0","Apila el valor <Null>");
   }
```

TipoInt.class

```
@Override
   public void gen(Token k) {
```

```
    GCI.gen().gen("PUSH " + k.getLexema(),"Apila el literal entero <" + k.getLexema()+">");
  }
```

TipoString.class

```
@Override
        public void gen(Token k) {
                GCI.gen().data();
                String lbl = GCI.gen().label();
                GCI.gen().gen("str_" + lbl,"DW " + '\"' + k.getLexema() +'\"' + ",0","");
                GCI.gen().code();
                GCI.gen().gen("PUSH str_" + lbl,"");
        }
```

CSystem.class

```
public void init() throws SemanticException {
   Clase clase = TS.ts().addClass(new Token("id", 0, "System"));
   clase.setAncestro(new Token("id", 0, "Object"));
   clase.setConstructor(new Token("id", 0, "System"));
   Metodo read = clase.addMetodo("static", new Token("id", 0, "read"), TipoInt.instance());
   Metodo printB = clase.addMetodo("static", new Token("id", 0, "printB"), TipoVoid.instance());
   Metodo printI = clase.addMetodo("static", new Token("id", 0, "printI"), TipoVoid.instance());
   Metodo printC = clase.addMetodo("static", new Token("id", 0, "printC"), TipoVoid.instance());
  Metodo printS = clase.addMetodo("static", new Token("id", 0, "printS"), TipoVoid.instance());
  Metodo println = clase.addMetodo("static", new Token("id", 0, "println"), TipoVoid.instance());
 Metodo printBln = clase.addMetodo("static", new Token("id", 0, "printBln"), TipoVoid.instance());
  Metodo printIln = clase.addMetodo("static", new Token("id", 0, "printIln"), TipoVoid.instance());
  Metodo printCln = clase.addMetodo("static", new Token("id", 0, "printCln"),TipoVoid.instance());
  Metodo printSln = clase.addMetodo("static", new Token("id", 0, "printSln"), TipoVoid.instance());

        printB.addArgFormal(new Token("id", 0, "b"), TipoBool.instance(), 0);
        printI.addArgFormal(new Token("id", 0, "i"), TipoInt.instance(), 0);
        printC.addArgFormal(new Token("id", 0, "c"), TipoChar.instance(), 0);
        printS.addArgFormal(new Token("id", 0, "s"), TipoString.instance(), 0);

        printBln.addArgFormal(new Token("id", 0, "b"), TipoBool.instance(), 0);
        printIln.addArgFormal(new Token("id", 0, "i"), TipoInt.instance(), 0);
        printCln.addArgFormal(new Token("id", 0, "c"), TipoChar.instance(), 0);
        printSln.addArgFormal(new Token("id", 0, "s"), TipoString.instance(), 0);

        String imp_read = "READ\nSTORE 3";
        String imp_printB = "LOAD 3\nBPRINT";
        String imp_printI = "LOAD 3\nIPRINT";
```

```
String imp_printC = "LOAD 3\nCPRINT";
String imp_printS = "LOAD 3\nSPRINT";
String imp_println = "PRNLN";
String imp_printBln = imp_printB + '\n' + imp_println;
String imp_printIln = imp_printI + '\n' + imp_println;
String imp_printCln = imp_printC + '\n' + imp_println;
String imp_printSln = imp_printS + '\n' + imp_println;

read.setBloque(new BloqueSystem(imp_read));
printB.setBloque(new BloqueSystem(imp_printB));
        printI.setBloque(new BloqueSystem(imp_printI));
printC.setBloque(new BloqueSystem(imp_printC));
printS.setBloque(new BloqueSystem(imp_printS));
println.setBloque(new BloqueSystem(imp_println));
printBln.setBloque(new BloqueSystem(imp_printBln));
        printIln.setBloque(new BloqueSystem(imp_printIln));
printCln.setBloque(new BloqueSystem(imp_printCln));
printSln.setBloque(new BloqueSystem(imp_printSln));
}
```

NodoId.class

```
public Tipo chequear(Metodo metodo) throws SemanticException {
…
if (siguiente != null) {
        …
        if (metodo.getVarsLocales().containsKey(t.getLexema())) { // ES UNA
                                                                  // VARLOCAL
        GCI.gen().gen("LOAD " + va.getOffset(),  "Cargo la variable local <" + va.getID() + ">");
        }
        else {
         …
        GCI.gen().gen("LOAD 3", "Apila la referencia a THIS el cual apunta a un objeto de la clase
        <" + metodo.getClaseDeclaracion().getClassID() + ">");
        GCI.gen().gen("LOADREF " + va.getOffset(), "Almacena el tope de la pila en la variable de
        instancia <" + va.getID() + ">");
        …
} else {
// si k.lex hace referencia a una variable local
        if (metodo.getVarsLocales().containsKey(t.getLexema())) {
        GCI.gen().gen("STORE " + va.getOffset(),"Almacena el tope de la pila en la variable local
        <"+ va.getID() + ">");
```

```
        …
        }
// si k.lex hace referencia a un argumento
        else if (metodo.getArgsFormales().containsKey(t.getLexema())) {
                GCI.gen().gen("STORE " + va.getOffset(),"Almacena el tope de la pila en el
        argumento <" + va.getID() + ">");
// si k.lex hace referencia a un atributo de instancia.
else if (metodo.getClaseDeclaracion().getAtributosInstancia().containsKey(t.getLexema())) {
        GCI.gen().gen("LOAD 3","Apila la referencia a THIS el cual apunta a un objeto de la clase
        <"+ metodo.getClaseDeclaracion().getClassID() + ">");
        GCI.gen().gen("SWAP","Invierte los argumentos, es necesario para ejecutar STOREREF");
         GCI.gen().gen("STOREREF " + va.getOffset(),"Almacena el tope de la pila en la variable de
        instancia <"+ va.getID() + ">");
                …
}
…
}
```

NodoIdEncadenado.class

```
public Tipo check(Metodo metodo, Clase c) throws SemanticException {
        if (c.getAtributosInstancia().containsKey(t.getLexema())) {
        …
                if (nodoSiguiente != null) {
                …
                GCI.gen().gen("LOADREF " + va.getOffset(),"Almacena el tope de la pila en la
                variable de instancia <"+va.getID()+">");
                …}
                else {
                …}
        else{
        GCI.gen().gen("SWAP","Invierte los argumentos, es necesario para ejecutar STOREREF");
        GCI.gen().gen("STOREREF " + va.getOffset(),"Almacena el tope de la pila en la variable de
        instancia <"+va.getID()+">");
        }
```

NodoExpUnaria.class

```
public Tipo check(Clase clase, Metodo metodo) throws SemanticException {
        Tipo tipoIzq = eIzq.check(clase, metodo);
        switch (operador.getLexema()) {
                case "-":
                case "+":
                if (tipoIzq instanceof TipoInt) {
                        switch (operador.getLexema()) {
                        case "-":
                                GCI.gci().writeln("NEG","");
                        }
```

```
                return TipoInt().instance();
            }
            throw new SemanticException("Linea: " + operador.getLine() + " El tipo de la
expresion debe ser int.");
            case "!":
            if (tipoIzq instanceof TipoBool) {
                GCI.gci().writeln("NOT");
                return TipoBool().instance();
            }
            throw new SemanticException("Linea: " + operador.getLine() + " El tipo de la
expresion debe ser boolean.");
        }
        return null;
        }
```

NodoExpresionBinaria.class

```
public Tipo check(Metodo metodo) throws SemanticException {
        Tipo tipoIzq = eIzq.check(metodo);
        Tipo tipoDer = eDer.check(metodo);
        switch (operador.getLexema()) {
            case "+":
            case "-":
            case "*":
            case "/":
            case "%":
            if (tipoIzq instanceof TipoInt && tipoDer instanceof TipoInt) {
                switch (operador.getLexema()) {
                case "+":
                    GCI.gen().gen("ADD", "");
                    break;
                case "-":
                    GCI.gen().gen("SUB", "");
                    break;
                case "*":
                    GCI.gen().gen("MUL", "");
                    break;
                case "/":
                    GCI.gen().gen("DIV", "");
                    break;
                case "%":
                    GCI.gen().gen("MOD", "");
                    break;
                }
                return TipoInt.instance();
            }
            throw new SemanticException(operador.getLine(), "El tipo " + tipoDer.toString() +
            " no conforma con el tipo "  + tipoIzq.toString() + ".");
```

```
case "&&":
case "||":
if (tipoIzq instanceof TipoBool && tipoDer instanceof TipoBool) {
	switch (operador.getLexema()) {
	case "&&":
		GCI.gen().gen("AND", "");
		break;
	case "||":
		GCI.gen().gen("OR", "");
		break;
	}
	return TipoBool.instance();
}
throw new SemanticException(operador.getLine(), "El tipo " + tipoDer.toString() +
" no conforma con el tipo "+ tipoIzq.toString() + ".");
case ">":
case "<":
case ">=":
case "<=":
if (tipoIzq instanceof TipoInt && tipoDer instanceof TipoInt) {
	switch (operador.getLexema()) {
	case ">":
		GCI.gen().gen("GT", "");
		break;
	case "<":
		GCI.gen().gen("LT", "");
		break;
	case ">=":
		GCI.gen().gen("GE", "");
		break;
	case "<=":
		GCI.gen().gen("LE", "");
		break;
	}
	return TipoBool.instance();
}
throw new SemanticException(operador.getLine(), "El tipo " + tipoDer.toString() +
" no conforma con el tipo "
	+ tipoIzq.toString() + ".");
case "==":
case "!=":
if (tipoIzq.conforma(tipoDer) || tipoDer.conforma(tipoIzq)) {
	switch (operador.getLexema()) {
	case "==":
		GCI.gen().gen("EQ", "");
		break;
	case "!=":
		GCI.gen().gen("NE", "");
```

```
                                break;
                        }
                        return TipoBool.instance();
                }
                throw new SemanticException(operador.getLine(), "El tipo " + tipoDer.toString() +
" no conforma con el tipo "
                        + tipoIzq.toString() + ".");
        }
        return null;

        }
```

NodoIf.class

```
public boolean check(Metodo metodo) throws SemanticException {
        String l1 = GCI.gen().label();
        String l2 = GCI.gen().label();
        GCI.gen().openCommentD("Inicia bloque IF-THEN-ELSE");
        Tipo tipoExp = e.check(metodo);
        if (!(tipoExp instanceof TipoBool))
                throw new SemanticException(e.getLine(),"El tipo de la expresion debe ser
boolean.");
        GCI.gen().gen("BF " + l1,"");
        boolean hayReturnIf = sIf.check(metodo);
        boolean hayReturnElse = false;
        if (sElse != null) {
                GCI.gen().gen("JUMP " + l2,"");
                GCI.gen().gen(l1,"NOP","");
                hayReturnElse = sElse.check(metodo);
                GCI.gen().gen(l2,"NOP","");
        } else
                GCI.gen().gen(l1,"NOP","");

        GCI.gen().closeCommentD("Fin bloque IF-THEN-ELSE");
        return hayReturnIf && hayReturnElse;
        }
```

NodoWhile.class

```
public boolean check(Metodo metodo) throws SemanticException {
        String l1 = GCI.gen().label();
        String l2 = GCI.gen().label();
        GCI.gen().openCommentD("Inicio bloque WHILE");
        GCI.gen().gen(l1,"NOP","");
        Tipo tipoExp = e.check(metodo);
```

```
if (!(tipoExp instanceof TipoBool))
        throw new SemanticException(e.getLine() , "Se esperaba que la expresion sea de
        tipo boolean y es de tipo "   + tipoExp.toString() + ".");
GCI.gen().gen("BF " + l2,"");
boolean toRet = s.check(metodo);
GCI.gen().gen("JUMP " + l1,"");
GCI.gen().gen(l2,"NOP","");
GCI.gen().closeCommentD("Fin bloque WHILE");
return toRet;
}
```

NodoFor.class

```
public boolean check(Metodo metodo) throws SemanticException {
    String l1 = GCI.gen().label();
    String l2 = GCI.gen().label();
    GCI.gen().openCommentD("Inicia bloque FOR.");
    a1.check(metodo);
    GCI.gen().gen(l1,"NOP","");
    Tipo tipoExp = e.check(metodo);
    if (!(tipoExp instanceof TipoBool))
        throw new SemanticException(e.getLine(), "El tipo de la expresion debe ser boolean.");
    GCI.gen().gen("BF " + l2,"");
    boolean toRet = s.check(metodo);
    a2.check(metodo);
    GCI.gen().gen("JUMP " + l1,"");
    GCI.gen().gen(l2 ,"NOP","");
    GCI.gen().closeCommentD("Fin bloque FOR.");
    return toRet;
    }
```

NodoReturn.class

```
public boolean check(Metodo metodoLlamador) throws SemanticException {
    if (e != null) { // return algo;
        Tipo tipoExp = e.check(metodoLlamador);
        // si algo no conforma con el tipo de la declaracion del metodo
        // entonces hay error.
        if (!tipoExp.conforma(metodoLlamador.getRetorno()))
            throw new SemanticException(e.getLine(), "El tipo " + tipoExp.toString() + " no
            conforma con el tipo "+ metodoLlamador.getRetorno().toString() + ".");
        // ret_val = cantidad de argumentos + 1 (Puntero retorno) + 1
        // (enlace dinamico) + 1 (para llegar al retorno)
        int ret_val = metodoLlamador.getArgsFormales().size() + 3;
        if (metodoLlamador.isDynamic())
        ret_val++; // para pasar el this.
```

```
                GCI.gen().gen(
                        "STORE " + ret_val,
                        "Almacena el tope de la pila en la variable de Retorno del metodo <"
                                + metodoLlamador.getClaseDeclaracion().getClassID() + "::" +
metodoLlamador.getID() + ">");
        } else
        // return;
        // si es una funcion hay error porque se debe retornar algo!
        if (!(metodoLlamador.getRetorno() instanceof TipoVoid))
                        throw new SemanticException(k.getLine(), "Se debe retornar un resultado
                        de tipo " + metodoLlamador.getRetorno().toString()
                        + ".");
        if (metodoLlamador.getVarsLocales().size() > 0)
                GCI.gen().gen(
                        "FMEM " + metodoLlamador.getVarsLocales().size(),
                                        "Libera de la memoria las variables locales del metodo <" +
                                        metodoLlamador.getClaseDeclaracion().getClassID() + "::"+
                                        metodoLlamador.getID() + ">");

        GCI.gen().gen("STOREFP", "Reestablece el contexto.");
        if (metodoLlamador.isDynamic())
                GCI.gen().gen(
                        "RET " + (metodoLlamador.getArgsFormales().size() + 1),
                        "Retorna liberando de la memoria los argumentos, y el THIS del metodo <"
                                + metodoLlamador.getClaseDeclaracion().getClassID() + "::" +
metodoLlamador.getID() + ">");
        else
                GCI.gen().gen(
                        "RET " + metodoLlamador.getArgsFormales().size(),
                                        "Retorna liberando de la memoria los argumentos del
                                        metodo <" +
                                        metodoLlamador.getClaseDeclaracion().getClassID()+ "::" +
                                        metodoLlamador.getID() + ">");
        return true;
        }
```

NodoPrimNew.class

```
public Tipo check(Metodo llamador) throws SemanticException {
        Clase claseConstruir = TS.ts().getClass(k);
        // la verificacion de nombre del constructor fue analizado en la primer
        // pasada.
        Metodo constructor = claseConstruir.getConstructor();

        /*****************************************/
        // VERIFICO QUE LOS ARGUMENTOS ACTUALES DEL CONSTRUCTOR CONFORMEN LOS
        // ARGUEMNTOS FORMALES DE LA DECLARACION
```

```
		// verifico que la cantidad de argumentos formales y actuales sea la
		// misma.
		if (constructor.getArgsFormales().size() != argsActuales.size())
			throw new SemanticException(k.getLine(),"No se encuentra el constructor " +
constructor.toString() + ".");

		GCI.gen().openCommentD("Inicia construccion de un objeto de la clase
<"+claseConstruir.getClassID()+">");
		GCI.gen().gen("RMEM 1","Reserva espacio para el retorno del constructor de la clase <"+
claseConstruir.getClassID()+">");
			GCI.gen().gen("PUSH " + claseConstruir.getLastOffsetAI(),"Apila el tamano de CIR
			de la clase <"+claseConstruir.getClassID()+">");
		GCI.gen().gen("PUSH lmalloc","Reserva espacio en la memoria heap para el CIR");
		GCI.gen().gen("CALL","Invoca a la rutina de malloc.");
		GCI.gen().gen("DUP","Duplica la direccion del CIR que se encuentra en el tope de la pila.");
		GCI.gen().gen("PUSH " + claseConstruir.getCode(),"Apila la etiqueta de la VT de la clase
<"+claseConstruir.getClassID()+">");
		GCI.gen().gen("STOREREF 0","");
		GCI.gen().gen("DUP","");
		// para cada argumento formal a del constructor hago:

		Argumento aFormal =null;
		NodoExpresion e =null;
		Tipo tipoExpresion=null;

		List<Argumento> argsFormales =constructor.getArgsFormalesOrdenados();
		for (int i = 0; i<argsFormales.size();i++){
			aFormal = argsFormales.get(i);
			e = argsActuales.get(i);

			tipoExpresion = e.check(llamador);
			GCI.gen().gen("SWAP","");
			if (!tipoExpresion.conforma(aFormal.getTipo()))
			throw new SemanticException(e.getLine() , "El tipo " + tipoExpresion.toString() + "
no conforma con el tipo "
					+ aFormal.getTipo().toString() + ".");
		}

		/*****************************************/

			GCI.gen().gen("PUSH " + claseConstruir.getConstructor().getCode(),"Apila la
			etiqueta del constructor de la clase <"+claseConstruir.getClassID()+">");
		GCI.gen().gen("CALL","Hace una llamada al constructor de la clase
<"+claseConstruir.getClassID()+">");
		if (listaLlamadas.getList().size() > 0)
			return listaLlamadas.check(claseConstruir, "dinamica", llamador,true);
```

```
        GCI.gen().closeCommentD("Fin de la construccion del objeto de la clase
<"+claseConstruir.getClassID()+">");
        return claseConstruir.getConstructor().getRetorno();
        }
```

NodoPrimThis.class

```
public Tipo check(Metodo metodo) throws SemanticException{
        if (metodo.isStatic())
                throw new SemanticException(k.getLine(),"No es posible hacer referencia a this en
un metodo estatico.");

        GCI.gen().gen("LOAD 3", "Apila el puntero a THIS de la clase
<"+metodo.getClaseDeclaracion().getClassID()+">");
        return metodo.getClaseDeclaracion().getConstructor().getRetorno();}
```

NodoIdDirecto.class

```
public Tipo check( Metodo metodo) throws SemanticException {
…
// el id es una variable local
if (metodo.getVarsLocales().containsKey(k.getLexema())) {
…
GCI.gen().gen("LOAD " + v.getOffset(),"Apilo el contenido de la variable local <"+v.getID()+">");
}
// el id es un argumento
else if (metodo.getArgsFormales().containsKey(k.getLexema())) {
…
GCI.gen().gen("LOAD " + v.getOffset(), "Apilo el contenido del argumento <"+v.getID()+">");
}
…
// el id es un atributo de instancia.
else if ((metodo.isDynamic())&&
metodo.getClaseDeclaracion().getAtributosInstancia().containsKey(k.getLexema())) {
GCI.gen().gen("LOAD 3","Apilo la referencia a THIS el cual apunta a un objeto de la clase
<"+metodo.getClaseDeclaracion().getClassID()+">");
GCI.gen().gen("LOADREF " + v.getOffset(),"Apilo el contenido de la variable de instancia
<"+v.getID()+">");
}
…
}
```

NodoIdEncadenadoDer.class

```
public Tipo chequear(Clase c,Clase claseActual,String tipo_llamada, Metodo metodo,Metodo
metodoActual,boolean flag) throws SemanticException {
…
// el id es un atributo de instancia.
 if ( c.getAtributosInstancia().containsKey(k.getLexema())) {
```

```
…
GCI.gen().gen("LOADREF " + v.getOffset(),"Apilo el contenido de la variable de instancia
<"+v.getID()+">");
…
}
…
}
```

NodoLlamadaDirecta.class

```
public Tipo check(Metodo metodo) throws SemanticException {
…
if(esteMetodo.isStatic()){
        genStatic(metodo, esteMetodo, claseActual);
        …}
else{
        genDynamic(metodo, esteMetodo, claseActual,flag);
        …}

private void genStatic(Metodo llamador, Metodo metodo, Clase objetoReceptor) throws
SemanticException {
if(!(metodo.getRetorno() instanceof TipoVoid))
        GCI.gen().gen("RMEM 1","Se reserva espacio para el retorno de la llamada al metodo
<"+metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
…
        GCI.gen().gen("PUSH " +metodo.getCode(),"Apila la etiqueta del metodo <"+
metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
        GCI.gen().gen("CALL","Hace la llamada al metodo
<"+metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
}

private void genDynamic(Metodo llamador, Metodo metodo, Clase objetoReceptor,boolean flag)
throws SemanticException {
if (llamador.isDynamic() && flag)
GCI.gen().gen("LOAD 3","Apila el puntero a THIS el cual apunta a un objeto de la clase <" +
llamador.getClaseDeclaracion().getClassID() +">");
if(!(metodo.getRetorno() instanceof TipoVoid)){
        GCI.gen().gen("RMEM 1","Se reserva espacio para el retorno de la llamada al metodo <"+
metodo.getClaseDeclaracion().getClassID()+ "::"+metodo.getID()+">");
        GCI.gen().gen("SWAP",""); // se agrega este swap por el this
        }
List<Argumento> argsFormales =metodo.getArgsFormalesOrdenados();
for (int i = 0; i<argsFormales.size();i++){
        …
        GCI.gen().gen("SWAP","");
        …
}
```

```
GCI.gen().gen("DUP","");
GCI.gen().gen("LOADREF 0","Accede a la VT de la clase
<"+metodo.getClaseDeclaracion().getClassID()+">");
GCI.gen().gen("LOADREF " + metodo.getOffset(),"Se desplaza en la VT y Carga el metodo
<"+metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
GCI.gen().gen("CALL","Hace una llamada al metodo <"+
metodo.getClaseDeclaracion().getClassID()+ "::"+metodo.getID()+">");
}
```

LlamadaEncadenada.class

```
private void genStatic(Metodo metodo, List<NodoExpresion>argumentos,Metodo
metodoActualPosta, Clase claseActualPosta) throws SemanticException {
if(!(metodo.getRetorno() instanceof TipoVoid))
        GCI.gen().gen("RMEM 1","Se reserva espacio para el retorno de la llamada al metodo <"+
metodo.getClaseDeclaracion().getClassID()+ "::"+ metodo.getID()+">");
controlarArgumentos(metodo,argumentos,metodoActualPosta, claseActualPosta);
        GCI.gen().gen("PUSH " +metodo.getCode(),"Apila la etiqueta del metodo <"+
metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
        GCI.gen().gen("CALL","Hace la llamada al metodo
<"+metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
        }

private void genDynamic(Metodo llamador, Metodo metodo,boolean flag) throws
SemanticException {
if(!(metodo.getRetorno() instanceof TipoVoid)){
                GCI.gen().gen("RMEM 1","Se reserva espacio para el retorno de la llamada al
metodo <"+ metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
                GCI.gen().gen("SWAP",""); // se agrega este swap por el this
        }
        // controlo que el tipo de lo argumentos actuales conforme los tipos
        // de los argumentos formales, ademas se genera el codigo.
                Argumento aFormal =null;
                NodoExpresion e =null;
                Tipo tipoExpresion=null;
                List<Argumento> argsFormales =metodo.getArgsFormalesOrdenados();
                for (int i = 0; i<argsFormales.size();i++){
                        aFormal = argsFormales.get(i);
                        e = argumentos.get(i);
                        tipoExpresion = e.check(llamador);
                        if (!tipoExpresion.conforma(aFormal.getTipo()))
                        throw new SemanticException(e.getLine() , "El tipo " +
tipoExpresion.toString() + " no conforma con el tipo "
                                        + aFormal.getTipo().toString() + ".");
                GCI.gen().gen("SWAP","");
                }
        GCI.gen().gen("DUP","");
```

```
        GCI.gen().gen("LOADREF 0","Accede a la VT de la clase
<"+metodo.getClaseDeclaracion().getClassID()+">");
        GCI.gen().gen("LOADREF " + metodo.getOffset(),"Se desplaza en la VT y Carga el metodo
<"+metodo.getClaseDeclaracion().getClassID()+ "::"+metodo.getID()+">");
        GCI.gen().gen("CALL","Hace una llamada al metodo
<"+metodo.getClaseDeclaracion().getClassID()+"::"+metodo.getID()+">");
        }
```

NodoSenSimple.class

```
public boolean check(Metodo metodo) throws SemanticException {
        Tipo tipo = e.check(metodo);
        if (! (tipo instanceof TipoVoid))
        GCI.gen().gen("POP" ,"se elimina el valor de retorno porque no se asigna a nada");
        return false;
        }
```

BloqueSystem.class

```
public boolean check(Metodo metodo) throws SemanticException {
        String[]  simp = implementacion.split("\n");
        for (String s : simp){
                GCI.gen().gen(s,"");
        }
        return true;
        }
```

GCI.class

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

```
public class GCI {
        private static GCI gen;
        public static String path;

        public static GCI gen() {
        if (gen == null)
                gen = new GCI();

        return gen;
        }
        private int e;
        private FileWriter f;
        private PrintWriter pw;
        private String tab = "\t\t";
        private String spaces, nop;

        private GCI() {
        setSpaces(25);
        e = 0;
```

```
try {
        f = new FileWriter(path);
        pw = new PrintWriter(f);
} catch (Exception e) {
        System.out.println("Archivo de salida invalido.");
}
}

public void setSpaces(int max) {
String s = "";
for (int i = 0; i < max; i++)
        s += " ";
String ss="";
for(int i=0;i<max/1.5;i++){
        ss += "-";
}
nop = ss;
spaces = s;
}
public String label() {
String l = "L" + e;
e++;
return l;
}
public void close() throws IOException {
this.f.close();
}
public void ln() {
this.pw.println("");
}
public void openCommentD(String c) {
ln();
gen(";"+nop,c);
}
public void closeCommentD(String c) {
gen(";"+nop,c);
}
public void comment(String c) {
this.pw.println("; " + c);
}
public void code() {
this.pw.println(".CODE");
}

public void data() {
this.pw.println(".DATA");
}
public void gen(String label, String code, String comment) {
```

```
        String s = "";
        if (!label.equals(""))
                s += label + ": ";

        if (!code.equals(""))
                s += calc_spaces(label) + code;

        if (!comment.equals(""))
                s += calc_spaces(code) + "; " + comment;
        this.pw.println(s);
        }
        public void gen(String code, String comment) {
        String s = "";
        if (!code.equals(""))
                s += spaces + code;

        if (!comment.equals(""))
                s += calc_spaces(code) + "; " + comment;
        this.pw.println(s);
        }
        private String calc_spaces(String d) {
        if (d.length() == 0)
                return spaces;
        int s = spaces.length() - (d.length() + 2);
        String ss = "";
        if (s > 0)
                ss = spaces.substring(0, s);
        return ss;
        }
}
```

Testing

Como parte de la estapa de testing, se incluye en el proyecto un conjunto de casos de test para comprobar el correcto funcionamiento del compilador desarrollado.
Se incluyen tests para:

- Verificar el análisis léxico.
- Verificar el análisis sintáctico.
- Verificar el análisis semántico.
- Verificar la generación de código.

A contiuación se muestra los resultados esperados de los casos de tests, que corresponden con la generacion de codigo.

LinkedSearchBinaryTree.java

```
Se insertaran los siguientes animales:
15 -> Oso
45 -> Perro
56 -> Elefante
1 -> Koala
12 -> Leon
543 -> Tigre
156 -> Gato
34 -> Leopardo
26 -> Loro
11 -> Tucan
100 -> Pollo
320 -> Caballo
800 -> Vaca
810 -> Toro
901 -> Ardilla
43 -> Coyote
2 -> Lobo

Obtener: 543 > Tigre
Obtener: 11 > Tucan
Obtener: 2 > Lobo
Obtener: 810 > Toro
Obtener: 901 > Ardilla

Eliminar 543.
Eliminar 320.
Eliminar 901.

Obtener: 543 > No se ha encontrado.
Obtener: 901 > No se ha encontrado.
Obtener: 320 > No se ha encontrado.
Obtener: 43 > Coyote
Obtener: 12 > Leon


La ejecuci%n del programa finaliz% exitosamente.
```

Llamadas.java

```
Secuenciales
A(int)
A.m()
B(int)
B.m()
C(int)
C.m()
D(int)
D.m()
12345

Anidadas
A(int)
A.m()
B(int)
B.m()
C(int)
C.m()
D(int)
D.m()
54321


La ejecuci%n del programa finaliz% exitosamente.
```

Polimorfismo.java

Recursivos.java