Etapa 4

Compilador completo de Minijava + CeiVM

Compiladores e Intérpretes

Entrega: 28 de Noviembre

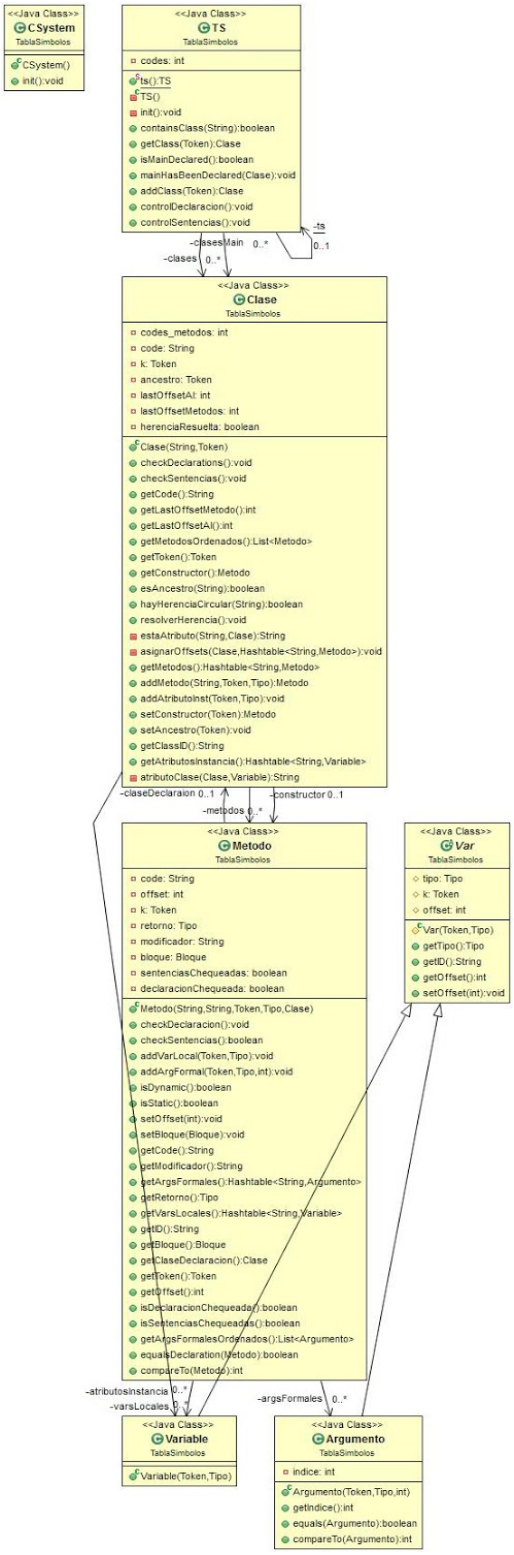
Integrantes:

Cuenca Francisco. LU:94294
Brenda Soledad Dilschneider. LU: 92774

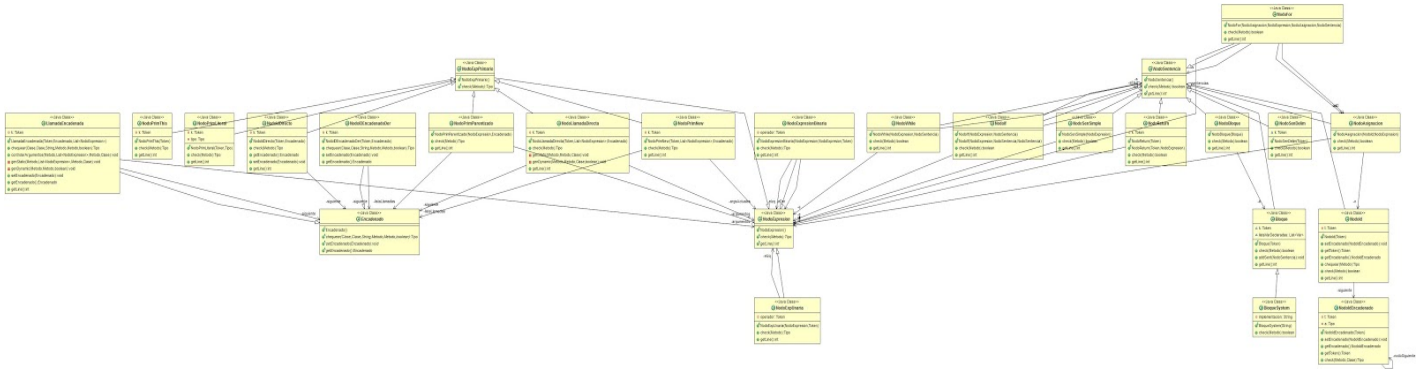
Diagrama de Clases.

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.

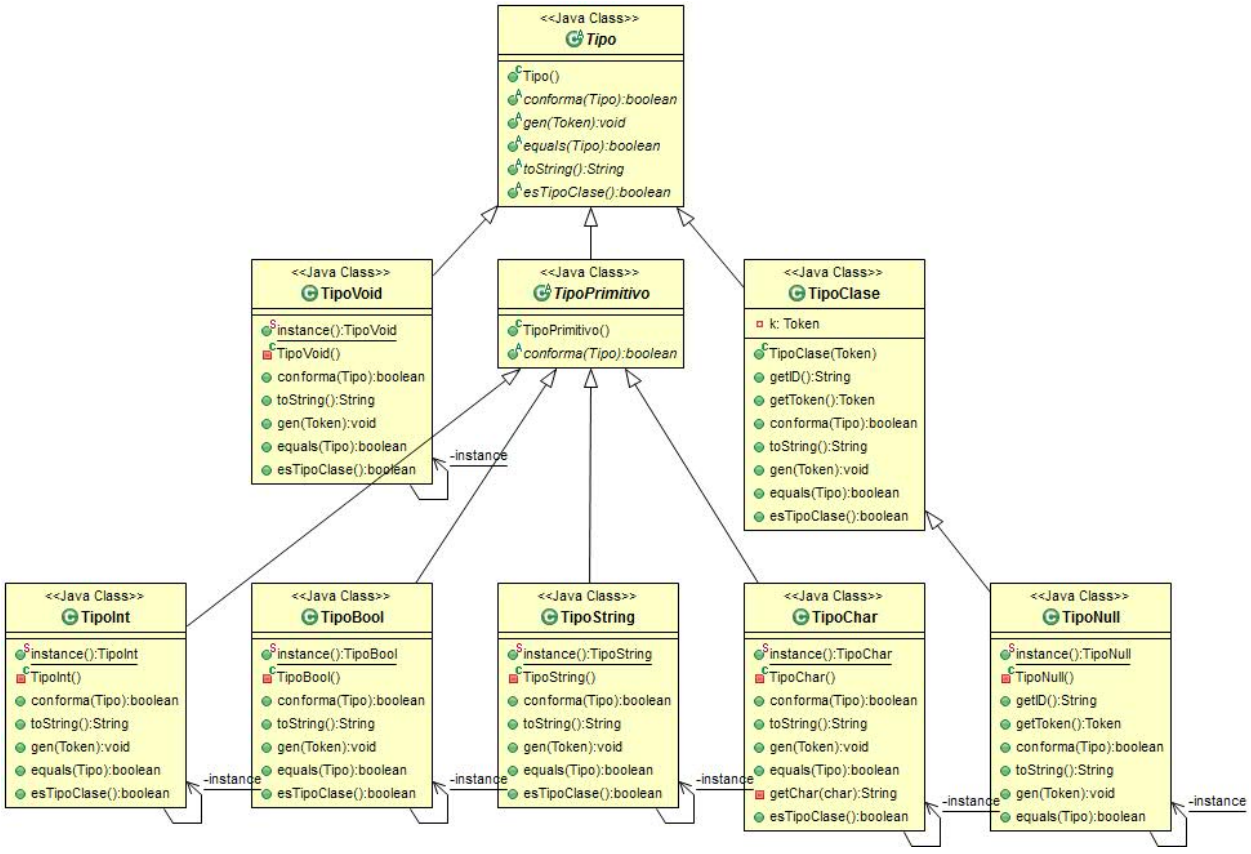
Diagramas de clases TS:



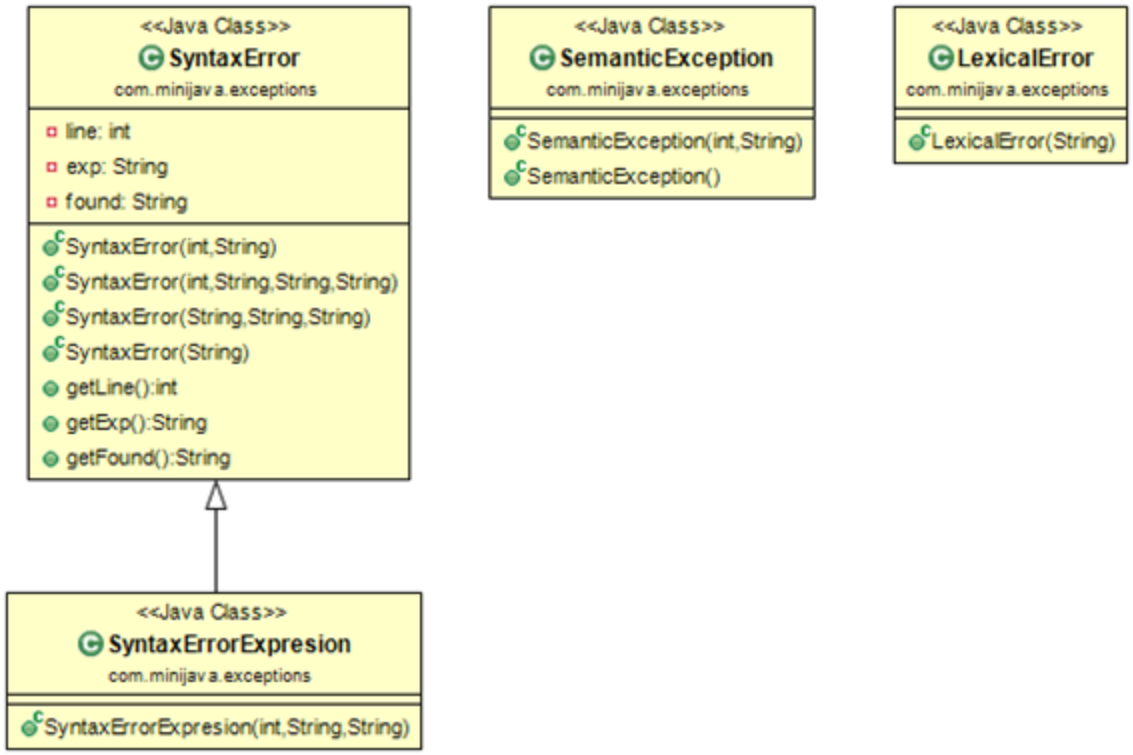
Diagramas de clases AST



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

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;

      }
```

Nodolf.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.

LinkedSearchBinaryTree.java

Invertir_numero.java

Llamadas.java

Polimorfismo.java

Recursivos.java

ListaEnlazada.java

```
Cantidad: 3
El valor del elemento de la lista es: 3
Pasa
El valor del elemento de la lista es: 2
Pasa
El valor del elemento de la lista es: 1
Pasa
Aca tendria que ir una excepcion pero devuelvo un nodo vacio
Se termino el programa exitodamente


La ejecuci¾n del programa finaliz¾ exitosamente.
```