Etapa 3

Analizador Semántico

Compiladores e Intérpretes

Entrega: 23 de Octubre

Integrantes: **Brenda Soledad Dilschneider. LU: 92774**

Cuenca Francisco. LU:94294

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> ;

<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>

<ListaDecVarsP> → , {
```
                  <ListaDecVars>.class = <ListaDecVarsP>.class
                  <ListaDecVars>.metodo = <ListaDecVarsP>.metodo
                  <ListaDecVars>.tipo = <ListaDecVarsP>.tipo
            }
```
<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)} $<SentenciaL>_1$

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

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

<IdEncadenados> → λ

<IdEncadenados> →. Identificador {<IdEncadenados>.lista_ids.agregarAlPrincipio(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>.llamadas = new List<NodoLlamada>()
} <LlamadaL> {<Primario>.prim= new NodoPrimLlamada1(<Expresion>.expr, <LlamadaL>.llamdas)}

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

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

<PrimarioP> → {<LlamadaL>.llamdas = new List<NodoLlamada>()} <LlamadaL> {
<PrimarioP>.prim = new NodoPrimLlamada2(<PrimarioP>.id, <LlamadaL>.llamadas)}

<PrimarioP> → <ArgsActuales> {<LlamadaL>.llamdas = new List<NodoLlamada>()} <LlamadaL> {
<PrimarioP>.prim = new NodoPrimLlamada3(<PrimarioP>.id, <ArgsActuales>.args,<LlamadaL>.llamadas)}

<LlamadaL> → <Llamada> {
<LlamadaL>.llamadas.add(<Llamada>.nodoLlamada)
<LlamadaL>1.llamadas = <LlamadaL>.llamadas
} <LlamadaL>$_1$

<LlamadaL> → λ

<Llamada> → .identificador <ArgsOpcionales> {<Llamada>.nodoLlamada = new NodoLlamada(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: lista_ids
Utilizado en: <LadoIzquierdo> , <IdEncadenados>
Tipo: NodoId
Descripcion: atributo sintetizado utilizado para almacenar identificadores encadenados.

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 la Tabla de Símbolos

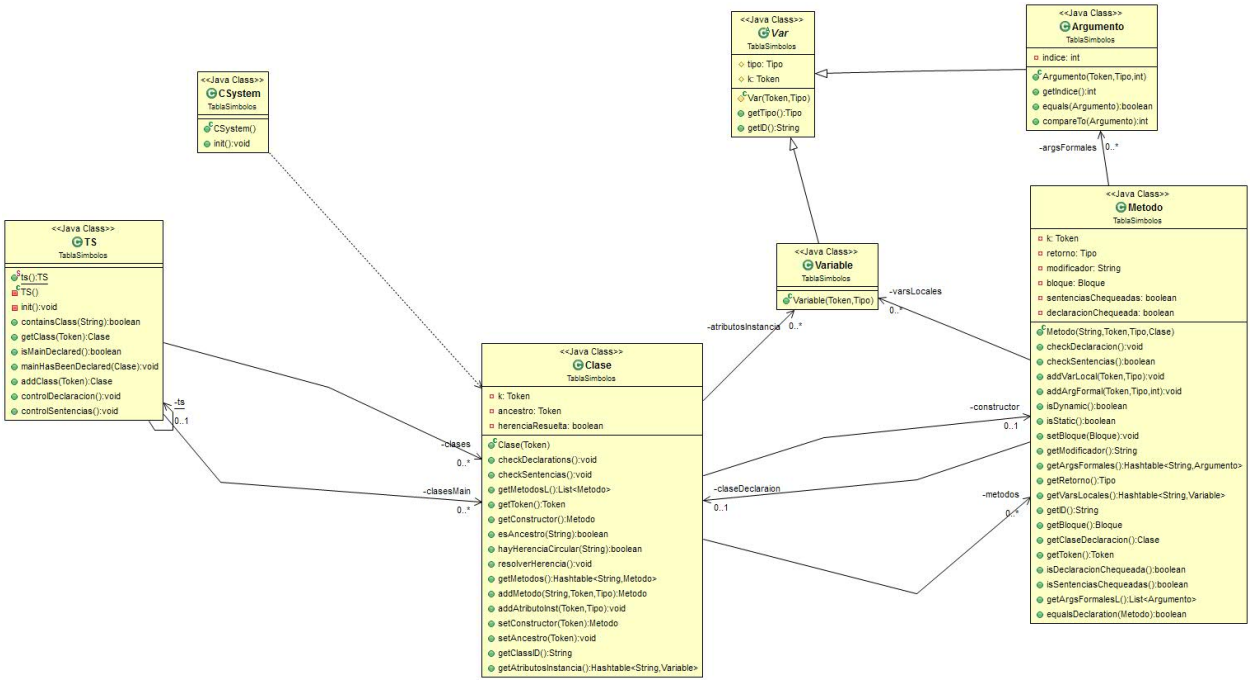


Diagrama del AST (Sentencias)

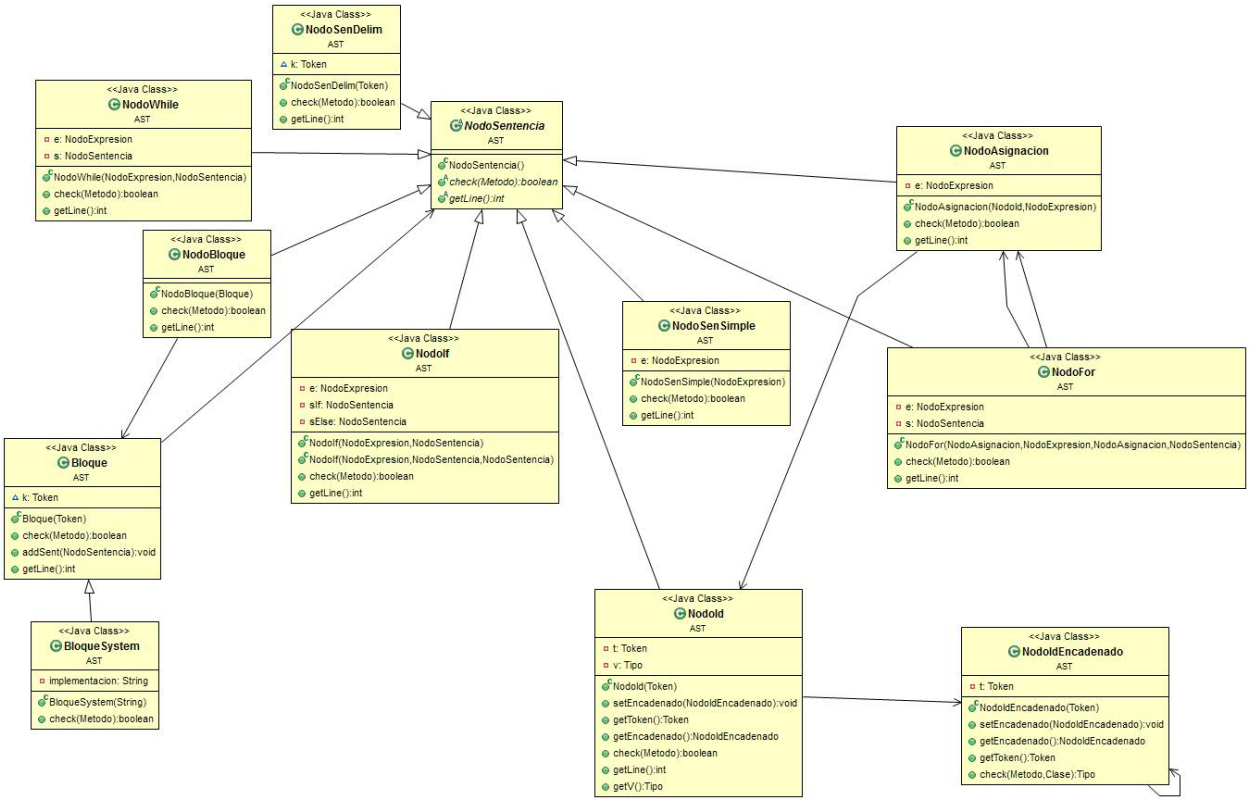
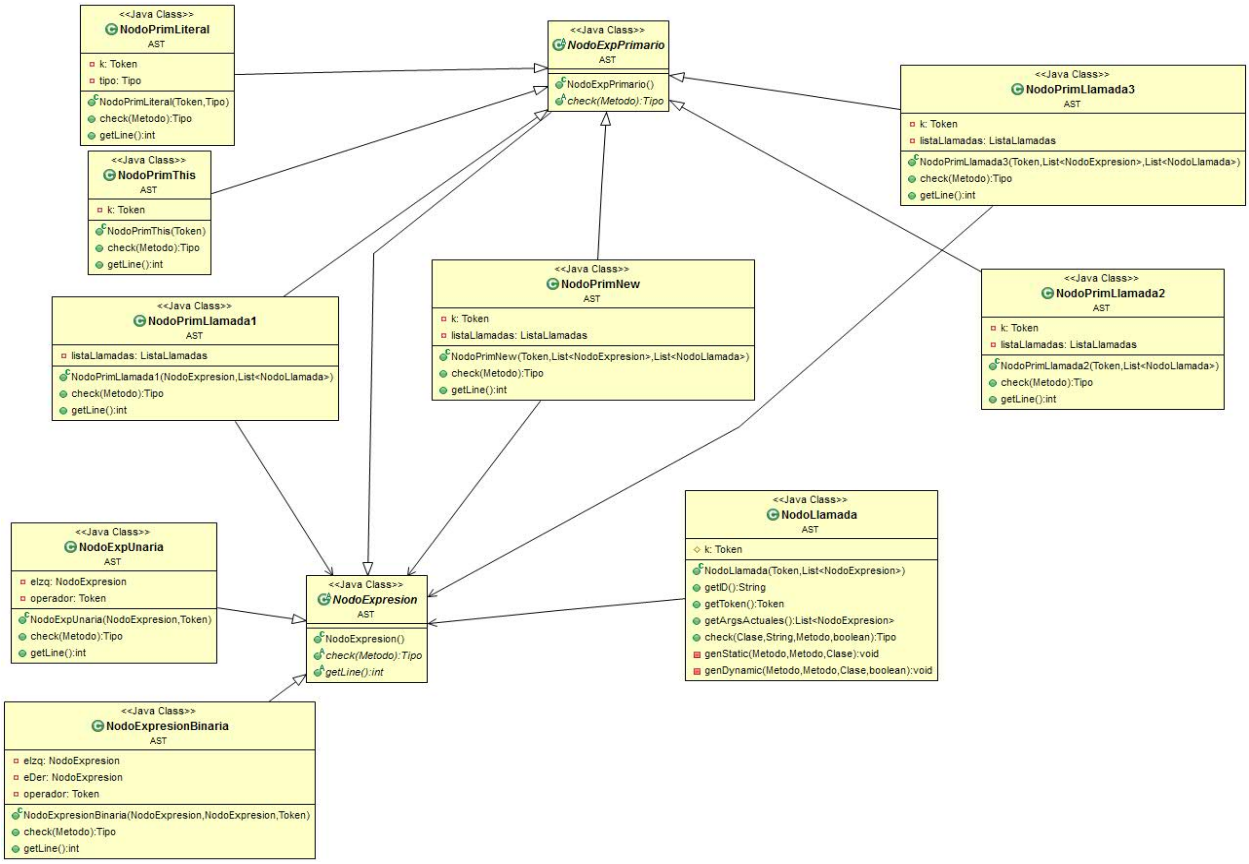
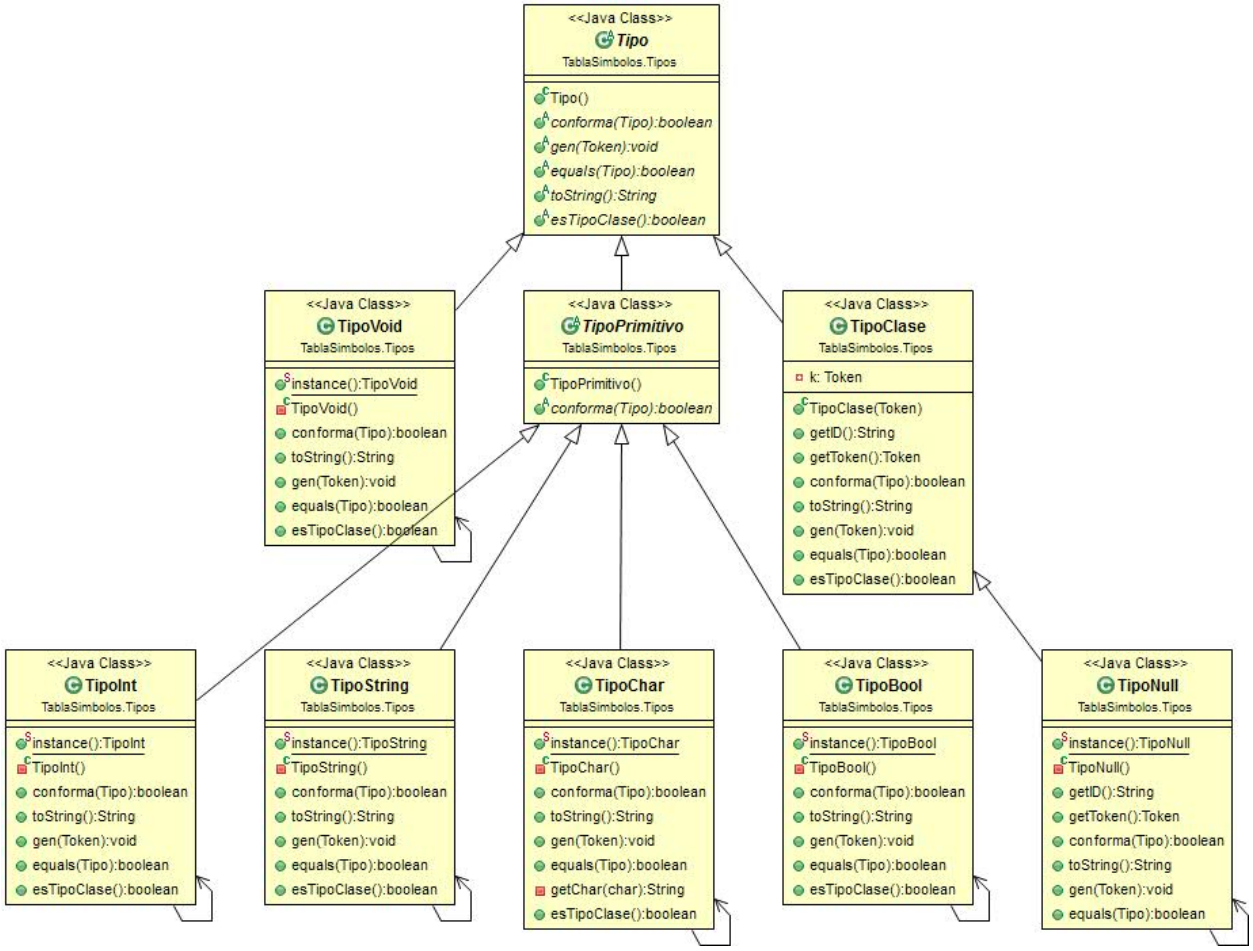


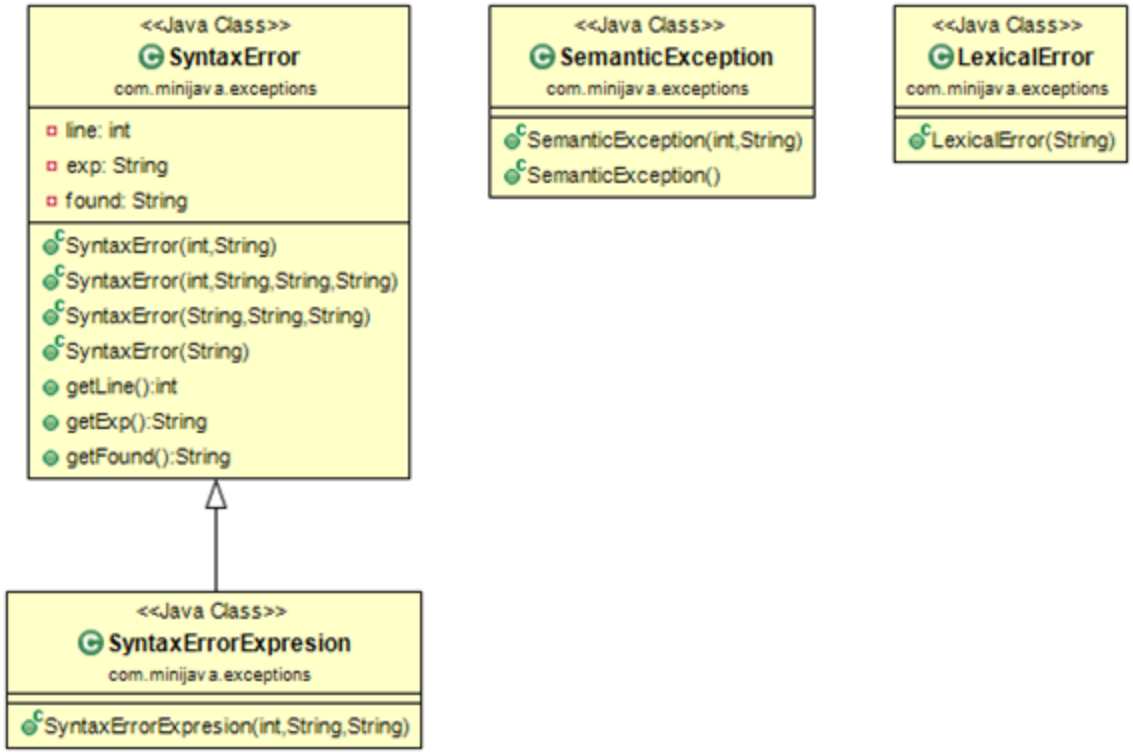
Diagrama del AST (Expresiones)



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

```
/**
	* Realiza el control de declaraciones.
	*
	* @throws SemanticException
	*/
	public void checkDeclarations() throws SemanticException {
		// la clase object esta perfecta
		if (this.getClassID().equals("Object"))
			return;

		// Checkeo las declaraciones de variables de instancia
		for (Variable v : atributosInstancia.values()) {
```

```
                if (v.getTipo() instanceof TipoClase)
                        TS.ts().getClass(((TipoClase) v.getTipo()).getToken());

        }

        // Controlo que no halla herencia circular y ademas controla que las
        // clases implicadas en la herencia esten declaradas.
        if (hayHerenciaCircular(this.getClassID()))
                throw new SemanticException(k.getLine(),
                                "Hay herencia circular en la clase " + this.getClassID());

        // Resuelvo la herencia de los metodos
        resolverHerencia();

        // Realizo el chequeo de declaracion de los metodos
        for (Metodo m : getMetodos().values())
                m.checkDeclaracion();

        // si esta clase tiene un metodo main() entonces le indica al TS que
        // esta clase tiene un metodo main.
        if (getMetodos().containsKey("main")
                        && getMetodos().get("main").getArgsFormales().size() == 0)
                TS.ts().mainHasBeenDeclared(this);

    }

/**
     * Realiza el control de sentencias.
     *
     * @throws SemanticException
     */
    public void checkSentencias() throws SemanticException {
        boolean hayReturn = constructor.checkSentencias(); // Chequea la

        // correctitud del

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

        // check del cuerpo de los metodos.
        for (Metodo m : getMetodosL()) {
```

```
                        if (!m.isSentenciasChequeadas()) {
                                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()
                                                                        + ".");

                        }
                }
        }
```

Metodo.class

```
/**
         * Realiza un chequedo de la declaracion del metodo.
         *
         * @throws SemanticException
         */
        public void checkDeclaracion() throws SemanticException {
                if (declaracionChequeada)
                        return;

                declaracionChequeada = true;

                // Si retorna un tipo clase, entonces dicha clase debe estar declarada.
                Tipo ret = getRetorno();
                if (ret instanceof TipoClase) {
                        if (!TS.ts().containsClass(((TipoClase) ret).getID()))
                                throw new SemanticException(((TipoClase) ret).getToken().getLine(),
"La clase " + ((TipoClase) ret).getID()
                                                + " no ha sido declarada.");
                }

                // Verifica que para los argumentos de tipo clase, la misma este
                // declarada.
                for (Argumento a : getArgsFormales().values()) {
                        if (a.getTipo() instanceof TipoClase) {
                                if (!TS.ts().containsClass(((TipoClase) a.getTipo()).getID()))
                                        throw new SemanticException(((TipoClase)
a.getTipo()).getToken().getLine(), "La clase "
```

```
                                                    + ((TipoClase) a.getTipo()).getID() + " no ha sido
declarada.");
                        }
                }

                // Verifica que para las variables locales de tipo clase, la misma este
                // declarada.
                for (Variable a : getVarsLocales().values()) {
                        if (a.getTipo() instanceof TipoClase) {
                                if (!TS.ts().containsClass(((TipoClase) a.getTipo()).getID()))
                                        throw new SemanticException(((TipoClase)
a.getTipo()).getToken().getLine(), "La clase "
                                                    + ((TipoClase) a.getTipo()).getID() + " no ha sido
declarada.");
                        }
                }
        }

        /***
         * Chequea las sentencias correspondientes al bloque de este metodo
         *
         * @return true si el retorno del bloque es correcto; false, en otro caso.
         * @throws SemanticException
         */
        public boolean checkSentencias() throws SemanticException {
        return getBloque().check(this);

        }
```

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.**

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.