

## 11 Anexos



### Anexo A– Manual de instalación

#### A.1 Requerimientos del sistema

Software necesario:

- Eclipse 3.4 o superior
- Java Development Kit (JDK) 8

#### A.2 Proceso de instalación

Dado que no existe repositorio que contenga los plugins desarrollados no es posible la instalación desde el Market de Eclipse.

Debe realizarse la instalación manual de los plugins. Esto consiste en copiar los ficheros plugEditor.jar y plugEditor2.jar en la carpeta “plugins” que se encuentra en la carpeta raíz de instalación del entorno de Eclipse que se quiera utilizar.

Otra opción es copiar los ficheros plugEditor.jar y plugEditor2.jar en la carpeta “dropins” que se encuentra en la carpeta raíz de instalación del entorno de Eclipse que se quiera utilizar.

Cualquiera de las dos opciones debería hacer funcionar los plugins sin ningún problema y en cualquier versión de Eclipse IDE superior a la 3.4.

Para comprobar que la instalación se ha realizado correctamente debemos ir dentro de Eclipse a la pestaña “Help-> About Eclipse” y pulsar el botón “Installation Details”. En la ventana que se abre accedemos a la pestaña “Plug-ins” y buscamos “plugeditor”. Deben aparecer ambos en la lista de la siguiente forma:

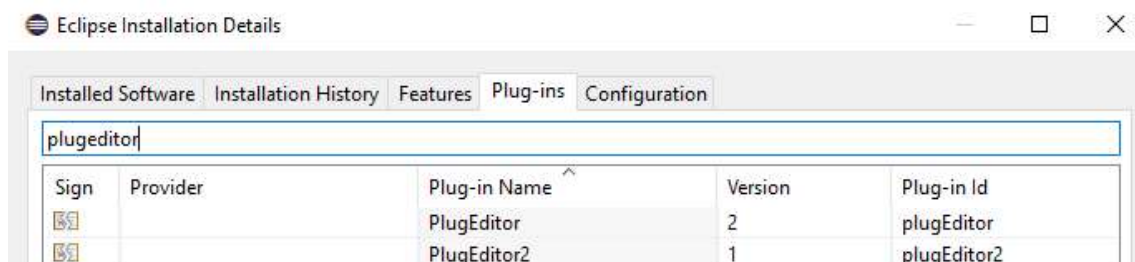


Ilustración 79 – Anexo A.2 – Proceso Instalación

## Anexo B – Manual de usuario

### B.1 PlugEditor (Procesadores de Lenguajes 1)

#### B.1.1 Ejecución

Para instalar el plugin se deben seguir los pasos del proceso de instalación detallados en el Anexo A.1.

La ejecución del plugin se realiza al dar doble clic sobre el fichero scanner.flex del proyecto del alumno que se encuentra en la carpeta “doc/specs”

#### B.1.2 Proceso de generación de tokens

The screenshot shows a window titled "Tokens" with a table and two buttons. The table has two columns: "Simbolo" and "Nombre". It contains a list of tokens and their corresponding names. To the right of the table are two buttons: "Añadir" and "Borrar".

Token	Nombre
(	OPPAR
)	CLPAR
*	MULT
,	COMA
-	MINUS
..	PUNTOS
:	DIV
::=	ASIG
;	PCOMA
=	EQUAL
>	MORE
ARRAY	ARRAY
BEGIN	BEGIN
BOOLEAN	BOOLEAN
CONST	CONST
DO	DO

*Ilustración 80 – Anexo B.1.2 – Generación de tokens*

Un token es una secuencia de caracteres que tiene significado en el lenguaje que va a desarrollarse. Se deben añadir todos los tokens del lenguaje a la lista correspondiente. No es necesario utilizar comillas para introducir el token, se añaden automáticamente a la declaración después.

## 11 - Anexos

Los tokens que van a coincidir con una expresión deben ir encerradas entre llaves como se muestra en la ilustración 81 y dicha expresión debe ser incluida en la lista de expresiones.

{identificador}	IDEN
{integer}	INT
{string}	STRING

*Ilustración 81 – Anexo B.1.2 – Token expresión*

### B.1.3 Proceso de generación de expresiones

Nombre	Expresion
fin	fin'{ESPACIO
identificador	[a-zA-Z]([a-z.,
integer	[0-9]+
nueva_linea	\\r\\n \\r\\n
string	\\\".*\\\"

*Ilustración 82 – Anexo B.1.3 – Generación de expresiones*

Se deben añadir las expresiones propias del lenguaje ingresando su nombre y patrón correspondiente. Se detallan los patrones más comunes:

- Dígito [0-9]
- Números 0|[1-9][0-9]\*
- Mayúsculas [A-Z]+
- Letra [a-zA-Z]
- Cadena \".\*\"
- Identificador [a-zA-Z]([a-zA-Z] | [0-9] ) \*

Es importante mencionar que una expresión puede formar parte de otra simplemente utilizando su nombre, así Identificador puede ser representado también como:

- Identificador Letra (Letra | Dígito) \*

### B.1.4 Proceso de generación de errores

Nombre	Mensaje
CaracterNoPer...	Error Lexico. ...
[^]	(error)lexicalE...

*Ilustración 83– Anexo B.1.4 – Generación de errores*

La inserción de errores léxicos es opcional dentro del lenguaje. Es importante que aparezca al menos uno “[^]”, cuyo significado es que acepta cualquier cadena. Los errores se sitúan al final del fichero scanner.flex por lo que son los últimos en leer, por lo tanto, en caso de llegar al patrón “[^]”, éste aceptará cualquier cadena indicando que no se ha encontrado coincidencia con los patrones declarados anteriormente.

Para la inserción de un error se deben rellenar dos campos:

- Nombre: Puede ser una expresión regular (patrón) o una expresión declarada en la lista de expresiones.
- Mensaje: Es el mensaje de error que se mostrará por consola al alumno en el caso de que se produzca dicho error.

### B.1.5 Declaración de comentarios

Inicio	Final
(*	*)

*Ilustración 84– Anexo B.1.5 – Declaración de comentarios*

Solo se contemplan los comentarios multilínea en este proyecto. Deben insertarse los símbolos que indican el inicio y final del comentario. Todo lo que se encuentre dentro será ignorado por el analizador.

### B.1.6 Generación del analizador léxico

Se recomienda realizar una copia de seguridad de los ficheros objeto de este proyecto (scanner.flex y parser.cup) antes de su utilización.

La generación del analizador se realiza al pulsar el botón “Generar” de la pestaña “Léxico”. Puede comprobarse su correcto funcionamiento una vez finalizado el guardado.

### B.1.7 Declaración de no terminales

Dentro de la pestaña de sintáctico encontramos todas las funcionalidades necesarias para la generación del analizador sintáctico.

*Ilustración 85– Anexo B.1.7 – Declaración de no terminales*

Se detallan las funcionalidades disponibles relativas a los no terminales:

- Limpiar: Deshecha los cambios realizados y limpia la pantalla
- Añadir No terminal: Crea un no terminal nuevo sin producciones con el nombre indicado
- Añadir Producción: Añade una nueva producción al no terminal que se encuentra en el campo de edición
- Guardar No Terminal: Guarda el no terminal que se encuentra en el campo de edición y limpia la pantalla
- Editar: Al seleccionar un no terminal y pulsar este botón el no terminal aparece en pantalla listo para la edición

- Eliminar No terminal: Al seleccionar un no terminal y pulsar este botón el no terminal es eliminado.

Además de las funcionalidades presentadas, en esta pestaña podemos encontrar las listas de tokens y no terminales declarados mostradas en la ilustración 86.

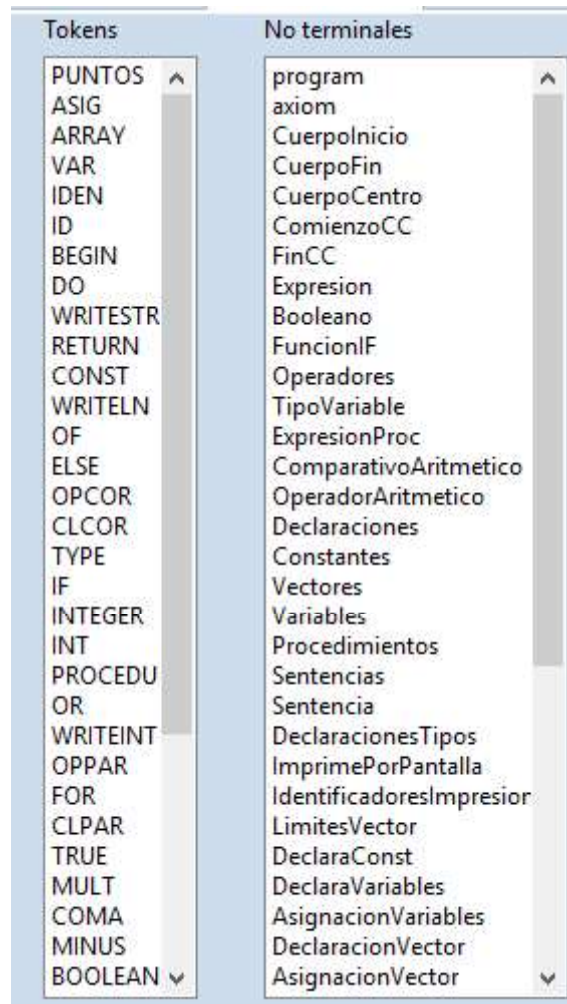


Ilustración 86 – Anexo B.1.7 – Listas de tokens y no terminales



### B.1.8 Generación del analizador sintáctico

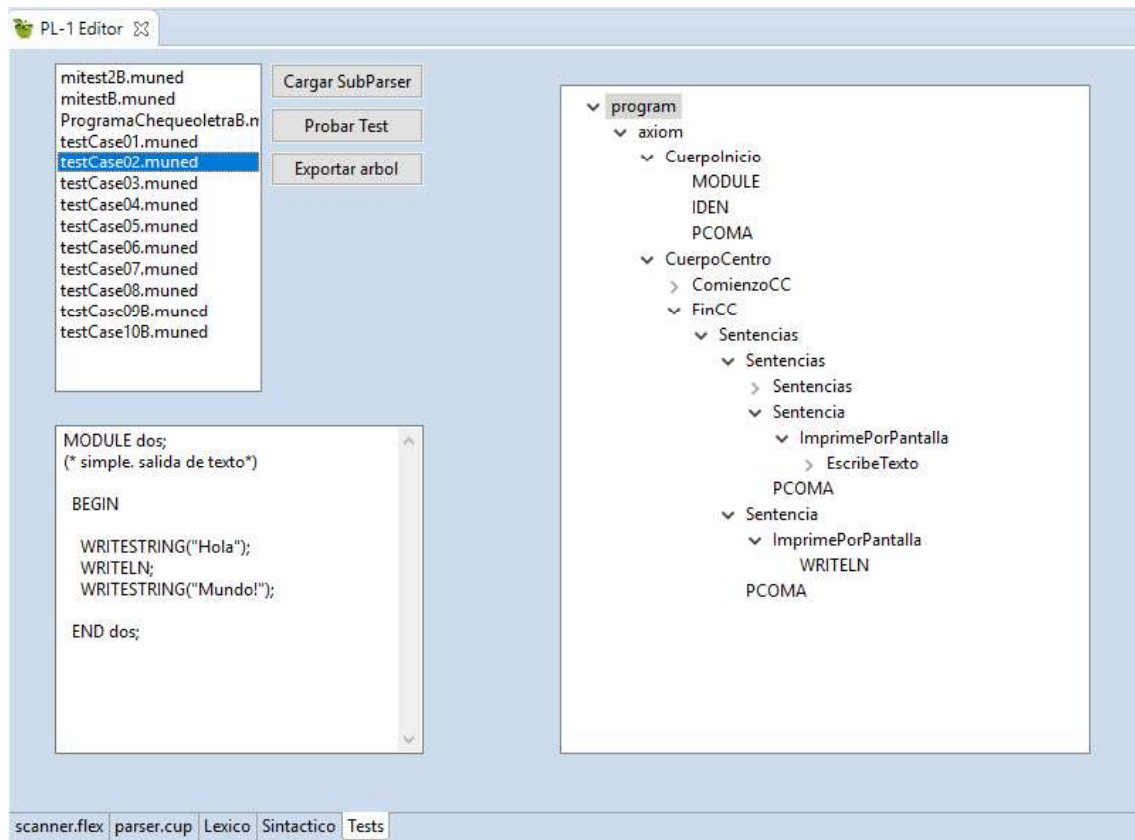
Antes de generar el analizador es importante comprobar que no se han producido errores con las modificaciones realizadas. Para tal fin sirve el botón “CHECK” que, aunque no cubre todos los posibles errores, sí que nos avisa de los más comunes. Se recomienda su utilización antes de guardar el contenido.

La generación del analizador se realiza al pulsar el botón “Guardar” de la pestaña “Sintáctico”. Puede comprobarse su correcto funcionamiento una vez finalizado el guardado.

Se detallan las funciones de los botones auxiliares en la pestaña de sintáctico:

- Actualizar: Realiza la lectura de los ficheros scanner.flex y parser.cup extrayendo los tokens y no terminales declarados para rellenar las listas.
- Guardar: Guarda el trabajo realizado en los ficheros correspondientes.
- Check: Realiza comprobaciones de errores comunes sobre el trabajo realizado relativas a no terminales duplicados o no declarados.

## B.1.9 Utilización de los test PL1



*Ilustración 87 – Anexo B.1.9 – Tests PL1*

La pestaña Tests es la encargada de las ultimas comprobaciones sobre la gramática generada. Para ello deben generarse ficheros de texto con el código fuente correspondiente al lenguaje que se quiera producir. Dichos ficheros se han de situar en la carpeta “doc/test” del proyecto del usuario. El plugin leerá los ficheros contenidos en dicha carpeta.

Se detallan las funcionalidades disponibles:

- Cargar SubParser: Genera una clase auxiliar para el proyecto que se encarga de ejecutar los test. Debe ingresarse la extensión de los ficheros test dentro de esta clase para su correcto funcionamiento como se muestra en la ilustración 88. Antes de realizar otras acciones esta clase debe ser ejecutada como una aplicación java (posee Main).

## 11 - Anexos

```
subparser.java  ✖
1 package compiler.syntax;
2
3+ import java.io.BufferedWriter;
4
5
6 public class subparser extends parser{
7     //Insertar aqui la extensión de los ficheros de tests
8     public static String extension=".muned";
9 }
```

*Ilustración 88 – Anexo B.1.9 – Extensión de los tests PL1*

- Probar Test: Ejecuta el test seleccionado generando el árbol sintáctico
- Exportar árbol: Genera un fichero de texto con el contenido del árbol sintáctico ya generado.

Al ejecutar la clase subparser podremos observar como realiza todos los test encontrados en la carpeta “doc/tests” y genera unos ficheros con la extensión “.debug” dentro de “doc/tests/debug”. Estos ficheros son los que contienen la información relativa a la construcción del árbol sintáctico.

## B.2 PlugEditor2 (Procesadores de Lenguajes 2)

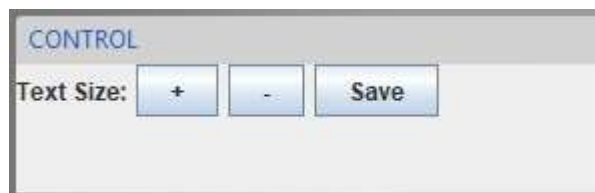
### B.2.1 Ejecución

Para instalar el plugin se deben seguir los pasos del proceso de instalación detallados en el Anexo A.1. La ejecución del plugin se realiza al dar doble clic sobre el fichero `parser.cup` del proyecto del alumno que se encuentra en la carpeta “doc/specs”.

### B.2.2 Generación del analizador semántico y código intermedio

La primera pestaña, `parser.cup`, contiene el código fuente (el contenido del fichero `parser.cup`) sobre el que se va a trabajar. En él pueden verse reflejados los cambios tras el guardado.

En la segunda pestaña, Semántico, podemos encontrar lo relativo a la creación del analizador semántico. Para su realización se debe partir de un lenguaje cuya estructura ya esté definida (el analizador sintáctico). Puede encontrarse el contenido de las acciones semánticas y de código intermedio dentro de los desplegables dedicados a separar los no terminales y sus producciones.

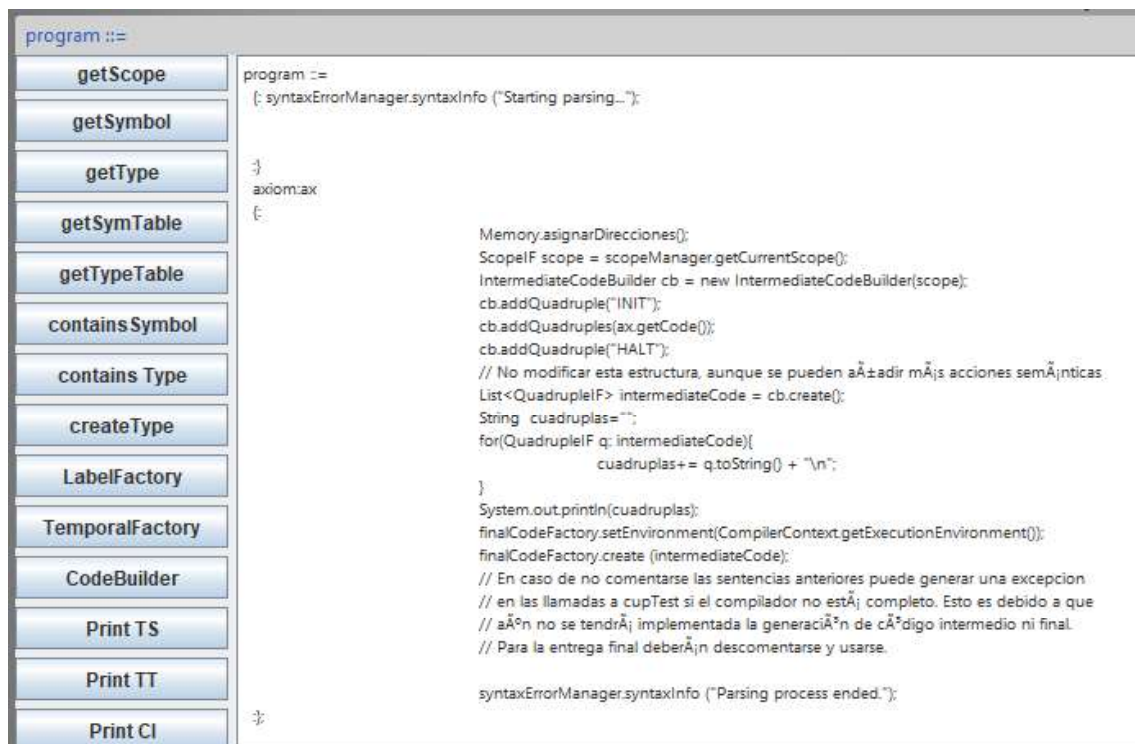


*Ilustración 89 – Anexo B.2.2 – Desplegable de control PL2-Semantico*

Las funcionalidades disponibles en el desplegable de control son:

- Aumentar tamaño del texto: Aumenta la fuente del texto en todos los desplegables
- Disminuir tamaño del texto: Disminuye la fuente del texto en todos los desplegables
- Save: Guarda los cambios realizados en “`parser.cup`”

## 11 - Anexos



*Ilustración 90 – Anexo B.2.2 – Desplegable con botones PL2-Semantico*

Además de las anteriores en cada desplegable se pueden encontrar botones arrastrables cuya finalidad es añadir código predefinido frecuente de manera rápida y fácil. La correspondencia botón-código es la que sigue:

- **getScope:**

```
ScopeIF scope = scopeManager.getCurrentScope();
```

- **getSymbol:**

```
SymbolIF symbol = scopeManager.searchSymbol(nombre);
```

- **getType:**

```
TypeIF tipo = scopeManager.searchType("Nombre");
```

- **getSymTable:**

```
SymbolTableIF symTable = scope.getSymbolTable();
```

- **getTypeTable:**

```
TypeTableIF typeTable = scope.getTypeTable();
```

- **containsSymbol:**

```
boolean contains=scopeManager.containsSymbol("Nombre");
```

- **contains Type:**

```
boolean contains=scopeManager.containsSymbol("Nombre");
```

- **createType:**

```
TypeSimple tsType = new TypeSimple(scopeManager.getCurrentScope(),  
"Nombre");  
scopeManager.getCurrentScope().getTypeTable().addType("Nombre",tsType);
```

- **LabelFactory:**

```
LabelFactory labelFactory = new LabelFactory();
```

- **TemporalFactory:**

```
TemporalFactory temporalFactory= new TemporalFactory(scope);
```

- **CodeBuilder:**

```
IntermediateCodeBuilder codeBuilder = new IntermediateCodeBuilder(scope);
```

- **Print TS:**

```
SymbolTableIF symTable = scope.getSymbolTable();  
List<SymbolIF> symList = symTable.getSymbols();  
for(SymbolIF sym:symList){System.out.println("Simbolo: "+sym+"\n");}
```

- **Print TT:**

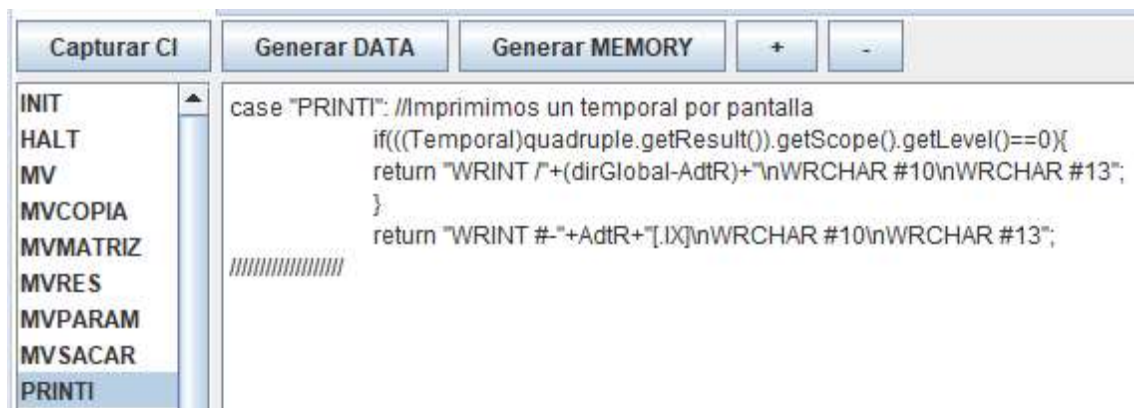
```
TypeTableIF typeTable = scope.getTypeTable();  
List<TypeIF> typeList = typeTable.getTypes();  
for(TypeIF type:typeList){System.out.println("Tipo: "+type+"\n");}
```

- **Print CI:**

```
ScopeIF scope = scopeManager.getCurrentScope();  
IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);  
List<QuadrupleIF> intermediateCode = cb.create();  
String cuadruplas="";  
for(QuadrupleIF q: intermediateCode){  
    cuadruplas+= q.toString() + "\n";  
}  
System.out.println(cuadruplas);
```

### B.2.3 Generación del código final

Para la elaboración del código final es necesario que se haya finalizado la generación de código intermedio y se hayan declarado las cuádruplas que van a ser utilizadas. A la izquierda encontraremos una lista con todas las cuádruplas declaradas.



*Ilustración 91 – Anexo B.2.3 – Generación de código final*

Se detallan las funcionalidades disponibles:

- Capturar CI: Realiza la lectura del fichero parser.cup y ExecutionEnvironmentEns2001.java en busca de cuádruplas o traducciones elaboradas con el fin de mostrarlas en una lista.
- Generar DATA: Elabora una clase java cuyo fin es ayudar a la asignación de posiciones de memoria para las cadenas de texto.
- Generar MEMORY: Elabora una clase java cuyo fin es ayudar a la asignación de posiciones de memoria para los símbolos y temporales pertenecientes al lenguaje.
- Aumentar tamaño del texto: Aumenta la fuente del texto en el área de texto
- Disminuir tamaño del texto: Disminuye la fuente del texto en el área de texto

## B.2.4 Utilización de los test PL2

La pestaña Tests es la encargada de las ultimas comprobaciones sobre la gramática generada. Para ello deben generarse ficheros de texto con el código fuente correspondiente al lenguaje que se quiera producir. Dichos ficheros se han de situar en la carpeta “doc/test” del proyecto del usuario. El plugin leerá los ficheros contenidos en dicha carpeta.

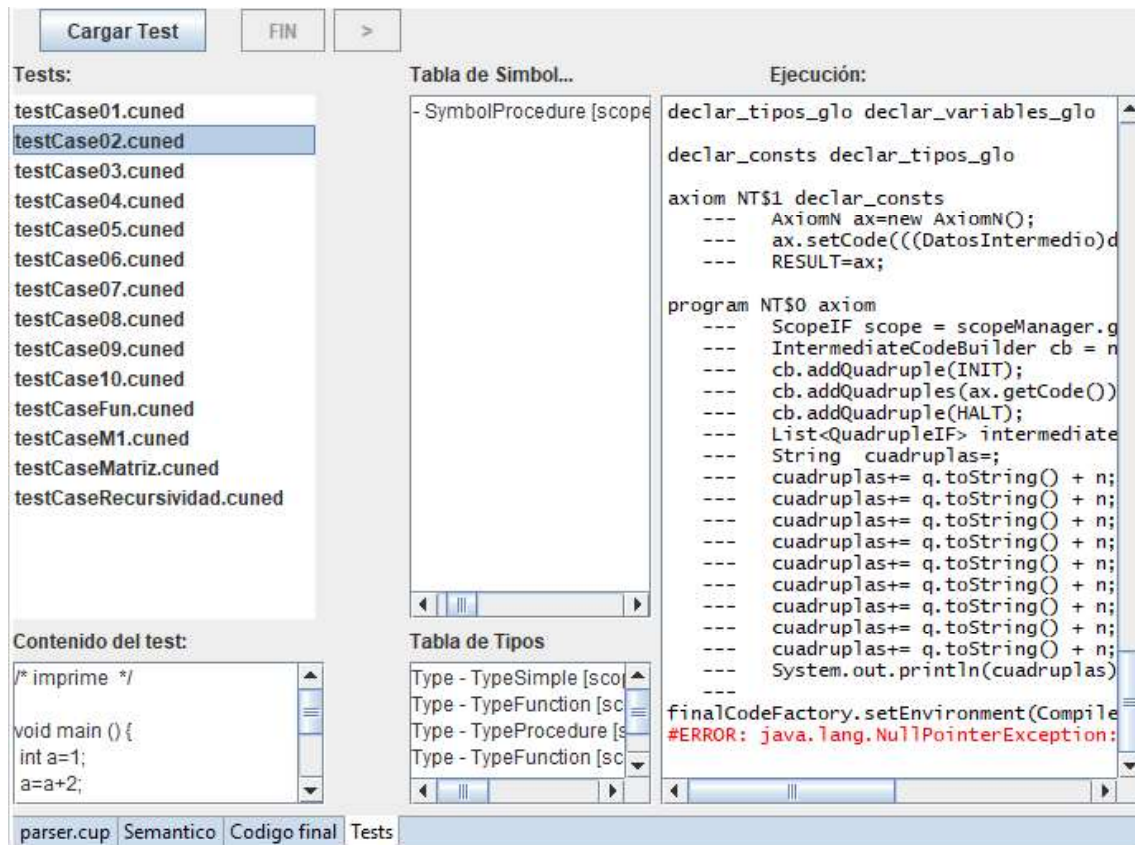


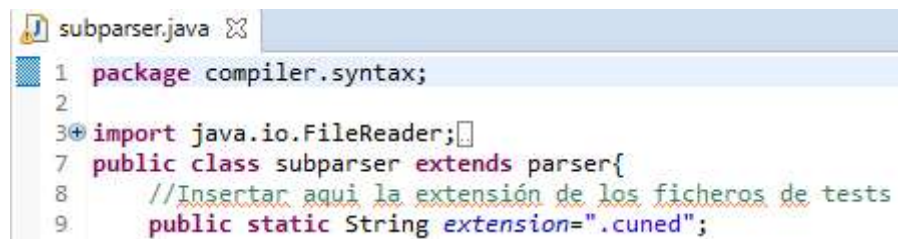
Ilustración 92 – Anexo B.2.4 – Tests en PL2

Se detallan las funcionalidades disponibles:

- Cargar Test: Genera una clase auxiliar, subparser.java, para el proyecto que se encarga de ejecutar los test. Debe ingresarse la extensión de los ficheros test dentro de esta clase para su correcto funcionamiento. Es necesario ejecutar el Main de dicha clase para la realización de los test, así como abrir el fichero parser.java para que se actualice su contenido.



## 11 - Anexos



```
1 package compiler.syntax;
2
3 import java.io.FileReader;
4
5 public class subparser extends parser{
6     //Insertar aqui la extensión de los ficheros de tests
7     public static String extension=".cuned";
8 }
```

*Ilustración 93 – Anexo B.2.4 – Extensión de los test PL2*

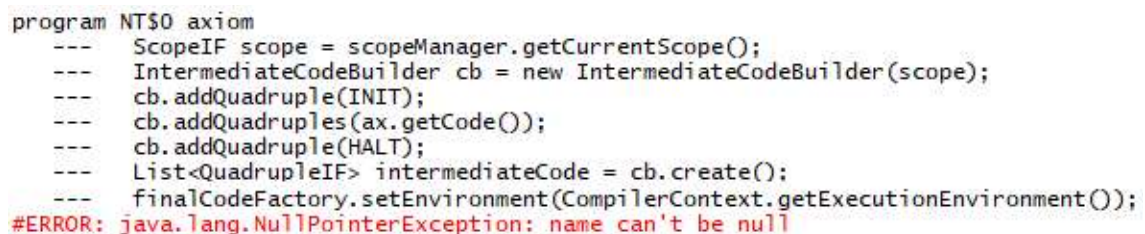
- FIN: Ejecuta el test mostrando la traza completa generada.
- Ejecución paso a paso (>): Avanza un paso en la ejecución del test actual.

Además de las funcionalidades mencionadas existen cuatro áreas de texto cuyo contenido es el siguiente:

- Contenido del test: Muestra el contenido del fichero de texto cargado como test
- Tabla de símbolos: Se actualiza a medida que avanza la ejecución del test y muestra el contenido de las tablas de símbolos.
- Tabla de tipos: Se actualiza a medida que avanza la ejecución del test y muestra el contenido de las tablas de tipos.
- Ejecución: Se actualiza a medida que avanza la ejecución del test y muestra las sentencias ejecutadas y el no terminal en el que se ejecutan.

Dentro del área de texto de la ejecución, mostrado en la ilustración 94, pueden producirse los siguientes casos:

- Texto pegado a la izquierda: Se corresponde con una producción de un no terminal
- Texto precedido por guiones (---): Se corresponde con una sentencia ejecutada
- Texto precedido por #ERROR en rojo: Se corresponde con un error producido en la ejecución.



```
program NT$0 axiom
--- ScopeIF scope = scopeManager.getCurrentScope();
--- IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);
--- cb.addQuadruple(INIT);
--- cb.addQuadruples(ax.getCode());
--- cb.addQuadruple(HALT);
--- List<QuadrupleIF> intermediateCode = cb.create();
--- finalCodeFactory.setEnvironment(CompilerContext.getExecutionEnvironment());
#ERROR: java.lang.NullPointerException: name can't be null
```

*Ilustración 94 – Anexo B.2.4 – Realización del test.*