



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática
de modalidad específica

**Desarrollo de herramientas de depuración para
prácticas de Procesadores del Lenguaje**

MIROSLAV VLADIMIROV VLADIMIROV
Dirigido por: ALVARO RODRIGO YUSTE
Curso: 2018 (Junio)

RESUMEN

Un compilador es un programa informático que traduce un programa que ha sido escrito en un lenguaje de programación a un lenguaje común. Está compuesto por análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, generación de código final.

En este contexto, el proyecto “Desarrollo de herramientas de depuración para prácticas de Procesadores del Lenguaje” tiene como objetivo el diseño e implementación de herramientas que faciliten el desarrollo del compilador realizado en las asignaturas *Procesadores de Lenguajes* de tercer curso de la UNED (Universidad Nacional de Educación a Distancia).

En el presente proyecto se han desarrollado dos plugins de eclipse utilizando el plugin “WindowBuilder” para la interfaz de usuario y el lenguaje de programación JAVA para el back-end.

El proyecto se centra en poner facilidades a la hora del desarrollo del compilador y en la depuración de código.

PALABRAS CLAVE

Procesadores de lenguajes, compilador, depuración, JAVA, análisis léxico, análisis sintáctico, análisis semántico, código intermedio, código final, generación de código.

ABSTRACT

A compiler is computer software that transforms computer code written in one programming language into another programming language. It's composed by lexical analysis, parsing, semantic analysis, conversión of input programs to an intermediate representation, code optimization and code generation.

In this context, the project "Development of debugging tools for Language Processor practices" has the objective of designing and implementing tools that facilitate the development of the compiler made in the subjects *Processors of Languages* of the third year of the UNED (National University of Distance Education).

In the present project two eclipse plugins have been developed using the "WindowBuilder" plugin for the user interface and the JAVA programming language for the back-end.

The project focuses on putting facilities at the time of compiler development and debugging code.

KEYWORDS

Processors of languages, compiler, debugging, JAVA, lexical analysis, parsing, semantic analysis, intermediate code, final code, code generation.

Tabla de contenido

Desarrollo de herramientas de depuración para prácticas de Procesadores del Lenguaje.....	i
1 Introducción	1
1.1 Motivación y definición del problema.....	3
1.2 Alcance y objetivos	4
2 Trabajos relacionados.....	5
3 Especificación del proyecto	9
3.1 Descripción general	11
3.1.1 Descripción del modelo	11
3.2 Características principales.....	12
3.3 Fases de desarrollo	13
3.3.1 Investigación preliminar	13
3.3.2 Encuesta a alumnos	13
3.3.3 Especificación del alcance de la herramienta.....	23
3.3.4 Diseño de la herramienta.....	24
3.3.5 Temporización	24
4 Análisis del sistema	25
4.1 Descripción del sistema	27
4.1.1 Características de los usuarios	28
4.1.2 Funcionalidades	28
4.1.3 Restricciones generales	29
4.2 Casos de uso	30
4.2.1 Herramienta de elaboración de compilador	30
4.2.2 Análisis Léxico	31
4.2.3 Análisis sintáctico.....	32
4.2.4 Tests PL1	33
4.2.5 Análisis Semántico y código intermedio.....	34
4.2.6 Código final.....	35
4.2.7 Tests PL2	36

5	Diseño e implementación	37
5.1	Estudio y selección de tecnologías	39
5.1.1	JFlex	39
5.1.2	CUP (Construction of Useful Parsers)	40
5.1.3	Window Builder	40
5.1.4	Sistema operativo	40
5.1.5	Lenguaje de programación	41
5.2	Diagramas de clases	43
5.2.1	Diagrama de Clases Procesadores de Lenguajes 1	43
5.2.2	Diagrama de Clases Procesadores de Lenguajes 2	51
5.3	Interfaz de usuario	57
5.3.1	Procesadores de Lenguajes 1	57
5.3.2	Procesadores de Lenguajes 2	71
6	Implementación	87
6.1	Entorno de desarrollo	89
6.2	Detalles de implementación	89
6.2.1	Fuentes de información	89
6.2.2	CUP	89
6.2.3	Patrones	90
6.2.4	Plugin	90
6.2.5	WindowBuilder	90
6.3	Detalles de implementación	90
6.3.1	Diseño del plugin	90
6.3.2	Diseño de las interfaces	90
6.3.3	Diseño de los patrones	91
6.3.4	Tratamiento de errores	91
7	Evaluación y pruebas	93
7.1	Casos de prueba realizados	95
8	Conclusiones y trabajo futuro	111
8.1	Conclusiones	113

8.2	Trabajo futuro.....	113
8.2.1	Herramienta educativa.....	113
8.2.2	Generador rápido de compiladores	113
8.2.3	Nuevas funcionalidades	114
9	Glosario y lista de acrónimos.....	115
10	Bibliografía	119
11	Anexos.....	123
	Anexo A– Manual de instalación	125
	A.1 Requerimientos del sistema	125
	A.2 Proceso de instalación.....	125
	Anexo B – Manual de usuario.....	126
	B.1 PlugEditor (Procesadores de Lenguajes 1).....	126
	B.1.1 Ejecución	126
	B.1.2 Proceso de generación de tokens	126
	B.1.3 Proceso de generación de expresiones.....	127
	B.1.4 Proceso de generación de errores	128
	B.1.5 Declaración de comentarios	128
	B.1.6 Generación del analizador léxico	129
	B.1.7 Declaración de no terminales	129
	B.1.8 Generación del analizador sintáctico.....	131
	B.1.9 Utilización de los test PL1	132
	B.2 PlugEditor2 (Procesadores de Lenguajes 2).....	134
	B.2.1 Ejecución	134
	B.2.2 Generación del analizador semántico y código intermedio	134
	B.2.3 Generación del código final	137
	B.2.4 Utilización de los test PL2	138

ÍNDICE DE ILUSTRACIONES

Ilustración 1 – CUP Eclipse Plugin texto	8
Ilustración 2 – CUP Eclipse Plugin Autómata	8
Ilustración 3 – Diagrama ¿Qué os ha resultado más difícil?	14
Ilustración 4 – Diagrama Dificultad de la instalación de la arquitectura ...	14
Ilustración 5 – Diagrama Tiempo empleado en el análisis léxico	15
Ilustración 6 – Comentarios Dificultades y carencias análisis léxico	15
Ilustración 7 – Diagrama Tiempo empleado en el análisis sintáctico	16
Ilustración 8 – Comentarios Dificultades y carencias análisis sintáctico ...	16
Ilustración 9 – Diagrama Tiempo empleado en el análisis semántico	17
Ilustración 10 – Comentarios Dificultades y carencias análisis semántico	18
Ilustración 11 – Diagrama Tiempo empleado en el código intermedio	18
Ilustración 12 – Comentarios Dificultades y carencias código intermedio	19
Ilustración 13 – Diagrama Tiempo empleado en el código final	19
Ilustración 14 – Comentarios Dificultades y carencias código final	20
Ilustración 15 – Comentarios Mejoras en las pedis	21
Ilustración 16 – Comentarios Opinión de los enunciados	22
Ilustración 17 – Comentarios Opinión sobre errores	22
Ilustración 18 – Comentarios Otros comentarios	23
Ilustración 19 – Temporización del proyecto	24
Ilustración 20 – Caso de uso 0 – Sistema completo	30
Ilustración 21 – Caso de uso 1 – Léxico	31
Ilustración 22 – Caso de uso 2 - Sintáctico	32
Ilustración 23 – Caso de uso 3 – Tests PL1	33
Ilustración 24 – Caso de uso 4 – Semántico y código intermedio	34
Ilustración 25 – Caso de uso 5 – Código final	35
Ilustración 26 – Caso de uso 6 – Tests PL2	36
Ilustración 27 – Diagrama de clases – Paquetes PL1	43
Ilustración 28 – Diagrama de clases – PlugEditor	43
Ilustración 29 – Diagrama de clases – Léxico	44
Ilustración 30 – Diagrama de clases – Common	45
Ilustración 31 – Diagrama de clases – Sintáctico	46
Ilustración 32 – Diagrama de clases – Tests PL1	49
Ilustración 33 – Diagrama de clases – Paquetes PL2	51
Ilustración 34 – Diagrama de clases – Semántico	51
Ilustración 35 – Diagrama de clases – Common	52
Ilustración 36 – Diagrama de clases – Tests PL2	53

Ilustración 37 – Diagrama de clases – Sintáctico PL2	54
Ilustración 38 – Diagrama de clases – FinalCode	55
Ilustración 39 – Diagrama de clases – pl2.editors.....	56
Ilustración 40 – Ejecución del plugin PL1.....	57
Ilustración 41 – Interfaz PL1 – scanner.flex	58
Ilustración 42 – Interfaz PL1 – parser.cup.....	59
Ilustración 43 – Interfaz PL1 - Léxico	59
Ilustración 44 – Interfaz PL1 Léxico - Tokens	60
Ilustración 45 – Interfaz PL1 Léxico - Expresiones.....	61
Ilustración 46 – Interfaz PL1 Léxico – Errores	61
Ilustración 47 – Interfaz PL1 Léxico – Comentarios.....	62
Ilustración 48 – Interfaz PL1 Léxico – Botón de guardado	62
Ilustración 49 – Interfaz PL1 Sintáctico	63
Ilustración 50 – Interfaz PL1 Sintáctico – Tokens	63
Ilustración 51 – Interfaz PL1 Sintáctico – No terminales.....	64
Ilustración 52 – Interfaz PL1 Sintáctico – Funciones de edición	64
Ilustración 53 – Interfaz PL1 Sintáctico – Vista “solo ver”	65
Ilustración 54 – Interfaz PL1 Sintáctico – Vista “editar”	65
Ilustración 55 – Interfaz PL1 Sintáctico – Botones auxiliares	66
Ilustración 56 – Interfaz PL1 Tests	67
Ilustración 57 – Interfaz PL1 Tests – Lista de tests.....	67
Ilustración 58 – Interfaz PL1 Tests – Botones de acciones	68
Ilustración 59 – Tests PL1 generados.....	68
Ilustración 60 – Interfaz PL1 Tests – Contenido del test	69
Ilustración 61 – Interfaz PL1 Tests – Árbol sintáctico.....	69
Ilustración 62 – Interfaz PL1 Tests – Árbol sintáctico texto plano.....	70
Ilustración 63 – Ejecución PL2	71
Ilustración 64 – Interfaz PL2 – parser.cup.....	72
Ilustración 65 – Interfaz PL2 Semántico.....	73
Ilustración 66 – Interfaz PL2 Semántico – Botones de control.....	73
Ilustración 67 – Interfaz PL2 Semántico – Desplegables	74
Ilustración 68 – Interfaz PL2 Código final.....	79
Ilustración 69 – Interfaz PL2 Código final – Botones.....	80
Ilustración 70 – Interfaz PL2 Código final – Lista de elementos CI	81
Ilustración 71 – Interfaz PL2 Código final – Traducción	81
Ilustración 72 – Interfaz PL2 Tests	82
Ilustración 73 – Interfaz PL2 Tests – Botones	82
Ilustración 74 – Interfaz PL2 Tests – Lista de tests.....	83

Ilustración 75 – Interfaz PL2 Tests – Contenido del test	83
Ilustración 76 – Interfaz PL2 Tests – Tabla de símbolos	84
Ilustración 77 – Interfaz PL2 Tests – Tabla de tipos	84
Ilustración 78 – Interfaz PL2 Tests – Ejecución del test.....	85
Ilustración 79 – Anexo A.2 – Proceso Instalación	125
Ilustración 80 – Anexo B.1.2 – Generación de tokens.....	126
Ilustración 81 – Anexo B.1.2 – Token expresión	127
Ilustración 82 – Anexo B.1.3 – Generación de expresiones	127
Ilustración 83– Anexo B.1.4 – Generación de errores.....	128
Ilustración 84– Anexo B.1.5 – Declaración de comentarios	128
Ilustración 85– Anexo B.1.7 – Declaración de no terminales	129
Ilustración 86 – Anexo B.1.7 – Listas de tokens y no terminales.....	130
Ilustración 87 – Anexo B.1.9 – Tests PL1.....	132
Ilustración 88 – Anexo B.1.9 – Extensión de los tests PL1.....	133
Ilustración 89 – Anexo B.2.2 – Desplegable de control PL2-Semantico...	134
Ilustración 90 – Anexo B.2.2 – Desplegable con botones PL2-Semantico	135
Ilustración 91 – Anexo B.2.3 – Generación de código final	137
Ilustración 92 – Anexo B.2.4 – Tests en PL2.....	138
Ilustración 93 – Anexo B.2.4 – Extensión de los test PL2	139
Ilustración 94 – Anexo B.2.4 – Realización del test.....	139

INDICE DE CASOS DE PRUEBA

Caso de prueba 01 - Añadir Token.....	95
Caso de prueba 02 - Borrar Token.....	96
Caso de prueba 03 - Añadir expresión.....	96
Caso de prueba 04 - Borrar expresión.....	97
Caso de prueba 05 - Añadir error.....	97
Caso de prueba 06 - Borrar error.....	98
Caso de prueba 07 - Guardar contenido.....	98
Caso de prueba 08 - Añadir no terminal.....	99
Caso de prueba 09 - Borrar no terminal.....	99
Caso de prueba 10 - Guardar no terminal.....	100
Caso de prueba 11 - Editar no terminal.....	100
Caso de prueba 12 - Añadir producción.....	101
Caso de prueba 13 - Eliminar producción.....	101
Caso de prueba 14 - Añadir componentes a la producción.....	102
Caso de prueba 15 - Guardar contenido.....	102
Caso de prueba 16 - Comprobar errores.....	103
Caso de prueba 17 - Cargar SubParser.....	103
Caso de prueba 18 - Probar Test.....	104
Caso de prueba 19 - Exportar árbol sintáctico.....	104
Caso de prueba 20 - Aumentar el tamaño del texto.....	105
Caso de prueba 21 - Disminuir tamaño del texto.....	105
Caso de prueba 22 - Introducir código mediante botones.....	106
Caso de prueba 23 - Aumentar tamaño del texto.....	106
Caso de prueba 24 - Disminuir tamaño del texto.....	107
Caso de prueba 25 - Capturar código intermedio.....	107
Caso de prueba 26 - Generar DATA.....	108
Caso de prueba 27 - Generar MEMORY.....	108
Caso de prueba 28 - Cargar Test.....	109
Caso de prueba 29 - FIN (Recorre toda la ejecución).....	109
Caso de prueba 30 - Ejecución paso a paso.....	110

1 Introducción

1 - Introducción

En este capítulo se proporciona una introducción al problema y se definen los objetivos a cumplir con la realización del PFG.

1.1 Motivación y definición del problema.

Hoy en día existen infinidad de lenguajes de programación, algunos de propósito general y otros específicos para sistemas aislados. Surge por tanto la necesidad de formar a la gente en su desarrollo, que además aporta conocimientos sobre el funcionamiento interno de los lenguajes actuales de alto nivel.

A pesar de que el desarrollo de compiladores no es algo nuevo, es una tarea dura que requiere gran cantidad de tiempo, conocimientos técnicos concretos y manejo con soltura de un lenguaje de programación de alto nivel.

En el Grado de Ingeniería Informática de la Universidad Nacional de Educación a Distancia (UNED España) hay dos asignaturas dedicadas a los compiladores, Procesadores de Lenguajes I y Procesadores de Lenguajes II, centrándose la primera en el análisis léxico y análisis sintáctico y la segunda en el análisis semántico, código intermedio y generación de código final.

En las asignaturas mencionadas el desarrollo del compilador se realiza con JFlex, CUP y la arquitectura proporcionada por el equipo docente.

La principal motivación del presente proyecto es proporcionar una herramienta para el desarrollo de compiladores que trabaje junto con la arquitectura proporcionada y permita a los estudiantes aprender los conceptos teóricos sin emplear demasiado tiempo en el desarrollo, permitiendo además la depuración de código.

1.2 Alcance y objetivos

El objetivo de este proyecto es desarrollar una herramienta software que trabaje junto a la arquitectura proporcionada por el equipo docente de las asignaturas, que sea fácil de utilizar y que sirva para simplificar el proceso de desarrollo y depuración de las prácticas de las asignaturas de Procesadores de Lenguajes, haciendo más sencillo y rápido el aprendizaje de la asignatura.

Dado que el temario referente al compilador se imparte en dos asignaturas independientes se van a desarrollar dos plugins independientes:

- **Procesadores de Lenguajes 1:** Abarca el análisis léxico y sintáctico. Se desarrollará una herramienta para facilitar la realización de ambas partes, con interfaz gráfica de fácil uso. Se prestará especial atención a la depuración de errores.
- **Procesadores de Lenguajes 2:** Abarca el análisis semántico, la generación de código intermedio y la generación de código final. Se desarrollará una herramienta para estructurar y depurar mejor el código.

2 Trabajos relacionados

2 - Trabajos relacionados

Para el desarrollo de las prácticas de Procesadores de Lenguajes se utilizan las herramientas JFLEX y CUP. En la actualidad hay otras herramientas más utilizadas como ANTLR o JAVACC, por lo que no existen muchas herramientas que apoyen el uso de JFLEX. Tras la búsqueda de programas similares tan solo se ha encontrado uno:

- **CUP Eclipse Plugin¹**: sirve como un IDE para el desarrollo de analizadores basados en CUP. Está destinado a ayudar en el desarrollo de la gramática y la depuración del código de acción. Fue producido durante un curso de laboratorio por un grupo de estudiantes (B. Engeser, S. Pretscher, J. Roith en orden alfabético).

Pros:

- Resalta el texto según la función que cumple en el compilador
- Generación del autómata
- Depuración de errores

Contras:

- Útil solo para análisis léxico y sintáctico
- Dependiente de la estructura de cup
- No compatible con la arquitectura del equipo docente

Como puede observarse, el único plugin similar al proyecto que se presenta a continuación tan solo ayuda en el desarrollo de parte del compilador, dejando el análisis semántico, generación de código intermedio y generación de código final totalmente en manos de los alumnos. No es compatible con la arquitectura sobre la que se elabora el presente proyecto. Sigue siendo necesario aprender la estructura de JFLEX y CUP además de los conocimientos teóricos propios de las asignaturas.

¹ CUP Eclipse plugin [en línea], <http://www2.cs.tum.edu/projekte/cup/eclipse.php> [Consulta 20/02/18]

Las ilustraciones 1 y 2 muestran la interfaz gráfica de CUP Eclipse Plugin:

```

1  parser.cup
2  %token SEMI PLUS MINUS LMINUS TIMES LPAREN RPAREN
3  %token Integer
4  %non-terminal expr_list Integer
5  %precedence left PLUS MINUS
6  %precedence left TIMES
7  %precedence left LMINUS
8  %grammar rules
9  expr_list ::= expr_list expr SEMI { System.out.println(e); }
10             | expr SEMI { System.out.println(e); }
11             ;
12  expr ::= expr e1 PLUS expr e2 { RESULT = e1+e2; }
13         | expr e1 MINUS expr e2 { RESULT = e1-e2; }
14         | expr e1 TIMES expr e2 { RESULT = e1*e2; }
15         | MINUS expr e { RESULT = -e; }
16         %prec LMINUS
17         | LPAREN expr e RPAREN { RESULT = e; }
18         | NUMBER:n { RESULT = n; }
19         ;

```

Ilustración 1 – CUP Eclipse Plugin texto

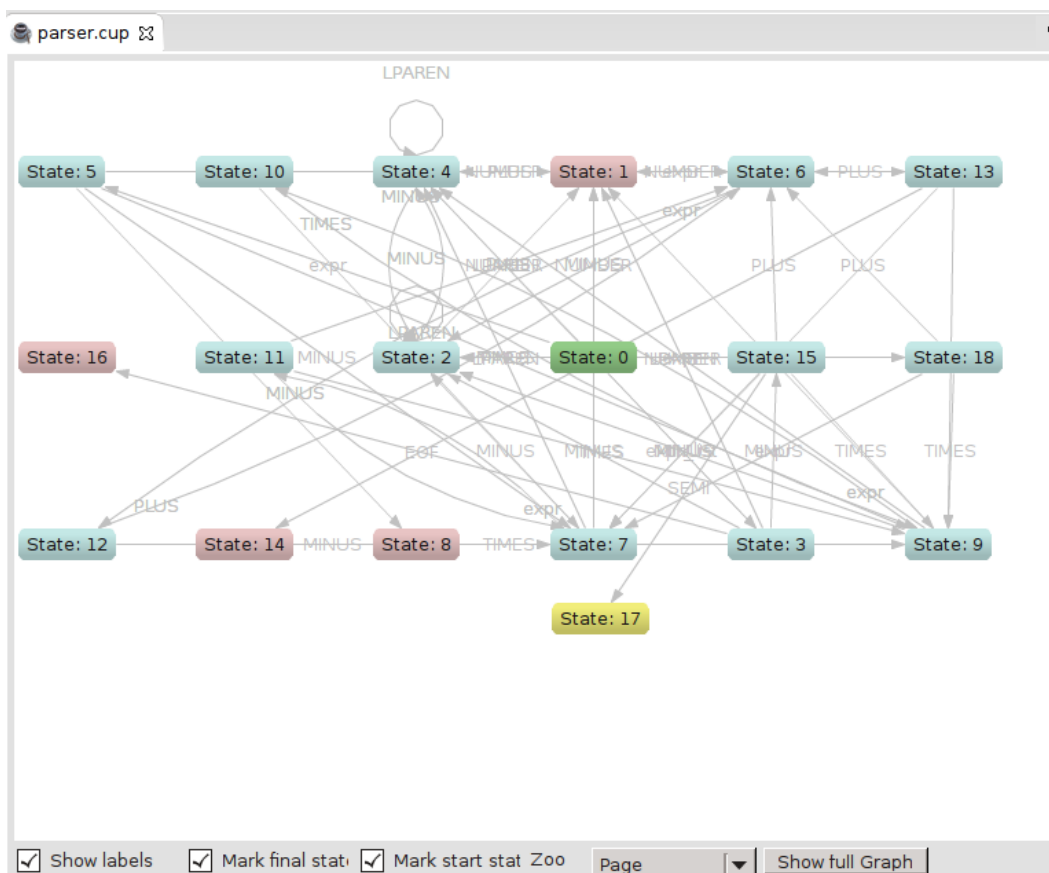


Ilustración 2 – CUP Eclipse Plugin Autómata

3 Especificación del proyecto

3 - Especificación del proyecto

En este capítulo se especifica cómo se ha acometido el proyecto. Se empieza dando una perspectiva general de la forma en la que se ha planteado conseguir los objetivos, describiendo el sistema.

3.1 Descripción general

Para llevar a cabo los objetivos descritos en el capítulo anterior se procede a construir dos aplicaciones independientes. La razón principal para separarlas está motivada por el hecho de que se imparten en dos asignaturas diferentes y aglutinar todas las funcionalidades en una sola herramienta probablemente entorpezca su uso. Cada parte del desarrollo del compilador tendrá su propia aplicación además de una parte para depuración del código en cada herramienta.

3.1.1 Descripción del modelo

Se ha valorado utilizar el modelo de desarrollo ágil, pero finalmente se ha optado por la utilización del modelo en iterativo.

Primero hay que explicar el desarrollo en cascada, cuyo enfoque metodológico ordena rigurosamente las etapas del proceso para el desarrollo del software, de tal forma que el inicio de cada etapa debe esperar a la finalización de la etapa anterior.

El desarrollo iterativo es un proceso de desarrollo de software creado en respuesta a las debilidades del modelo tradicional de cascada. Básicamente este modelo consiste en una serie de tareas agrupadas en pequeñas etapas repetitivas. Con esto se espera conseguir que las distintas partes puedan funcionar independientemente, y no se interrelacionen entre si más de lo necesario.

Por lo tanto, el desarrollo de la herramienta es igual que en el desarrollo del compilador donde no puede construirse la siguiente fase sin que la anterior esté acabada. Al finalizar cada etapa de desarrollo se comprobará el componente acabado antes de seguir con la siguiente.

3.2 Características principales

Teniendo en cuenta la poca variedad de herramientas de apoyo encontradas se ha perseguido dotar al programa de originalidad y funcionalidades que resulten adecuadas para los alumnos. Se enumeran características relevantes:

- El proyecto debe constar de 2 partes, una por asignatura
- Cada parte tendrá diferenciados sus componentes
- Debe ser visible el código original sobre el que se trabaja
- Se elaborará interfaz gráfica sencilla
- Es necesario buscar un punto intermedio entre automatizar el proceso de desarrollo del compilador y dejar que el alumno haga todo el trabajo. Lo ideal sería conseguir que se obtengan los mismos conocimientos técnicos con mucho menos trabajo y tiempo empleado.
- Hay que tener en cuenta la opinión de los alumnos, pues son los que más se beneficiarán de este proyecto.

3 - Especificación del proyecto

3.3 Fases de desarrollo

Como ya se ha mencionado, el método de desarrollo que se ha utilizado para cumplir los objetivos ha sido la realización de dos herramientas independientes de forma secuencial. Al desarrollo añadiremos también la investigación realizada y la generación de este documento.

3.3.1 Investigación preliminar

Antes de empezar se ha dedicado bastante tiempo a estudiar cuales serían las funciones que más ayudarían a los alumnos en la elaboración del compilador. Se ha realizado un estudio para determinar cuáles son los problemas encontrados, en que parte del compilador los encuentran y el tiempo correspondiente a la realización de cada parte. No obstante, la principal guía sobre el desarrollo del proyecto ha sido la propia, dado que también he tenido que realizar el desarrollo de un compilador.

Se ha investigado los distintos tipos de plugins que pudieran servir al proyecto y como implementarlos.

La mayor parte de la investigación se ha centrado en las propias herramientas JFLEX y CUP llegando a descompilar su código para entender el funcionamiento interno.

3.3.2 Encuesta a alumnos²

Se han utilizado los formularios de Google para poder recopilar información útil sobre los puntos en los que se emplea más tiempo. Para obtener la opinión de los alumnos sobre la dificultad del desarrollo del compilador, así como para entender cuáles han sido para ellos los puntos clave en los que han tenido más problemas, se ha realizado una encuesta anónima, que a pesar no tener un amplio número de respuestas ha servido para tener unas ideas claras sobre las dificultades. Se resumen a continuación los resultados obtenidos de las 13 respuestas.

²Encuesta realizada para el proyecto [en línea],
<https://docs.google.com/forms/d/1pR7uN3Mf6Ita4JSY5mXVBi-1rwF6c7T4-OICNEPeZBw/edit#responses>

3.3.2.1 ¿Qué os ha resultado más difícil?

Que os ha resultado mas difícil?

13 respuestas

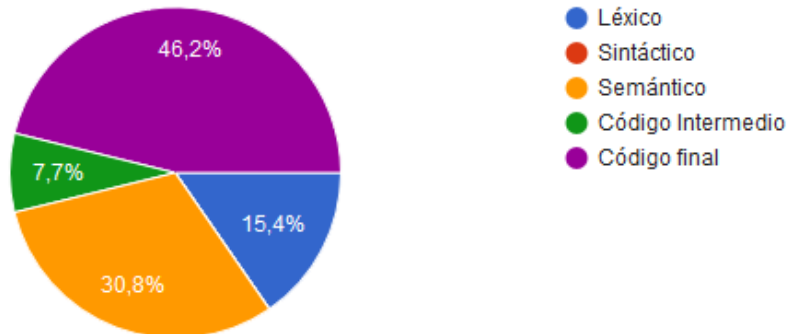


Ilustración 3 – Diagrama ¿Qué os ha resultado más difícil?

- Conclusión:

La mayoría de los alumnos coinciden en que la elaboración de código final es la más compleja en cuanto a conocimientos técnicos, seguida por poco por el análisis semántico, por lo que este proyecto debería centrarse en estas dos partes.

3.3.2.2 Dificultad de la instalación de la asignatura (0-10)

Dificultad de la instalación de la arquitectura

13 respuestas

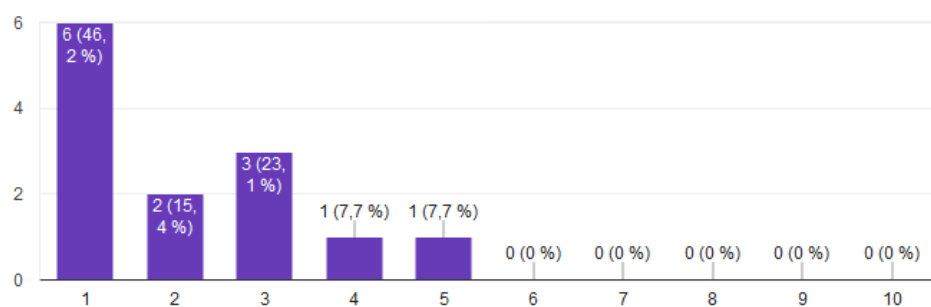


Ilustración 4 – Diagrama Dificultad de la instalación de la arquitectura

- Conclusión:

Al parecer la instalación de la arquitectura no requiere gran esfuerzo, además hay un video explicativo de cómo realizarla, por lo tanto, no será objetivo de este proyecto realizar mejoras en este aspecto.

3 - Especificación del proyecto

3.3.2.3 Tiempo empleado en el análisis léxico.

Cuanto tiempo consideras que te ha llevado la implementación del léxico?

13 respuestas

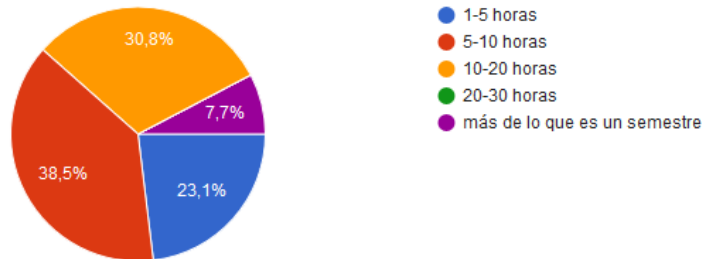


Ilustración 5 – Diagrama Tiempo empleado en el análisis léxico

- Conclusión:

Sin tener en cuenta el último comentario podemos concluir que la realización del análisis léxico lleva menos de 20 horas para la totalidad de los alumnos, por lo que se considera que esta parte no va a requerir un estudio en profundidad al no poder ahorrar demasiado tiempo a los alumnos.

3.3.2.4 Dificultades y carencias del análisis léxico

Que dificultades has tenido y que has echado en falta?

8 respuestas

Ejemplos resueltos
Entender un poco los estados. Haría falta más documentación sobre ello.
Hacen falta ejemplos de ello, sino es muy difícil empezar.
Nada.
ejemplos, documentacion
Más información de que había que hacer
Ninguna, nada
soporte a la depuración, enfoque de diseño

Ilustración 6 – Comentarios Dificultades y carencias análisis léxico

- Conclusión:

No es objetivo de este proyecto realizar documentación auxiliar para que los alumnos empleen más tiempo leyendo antes de comenzar a realizar el analizador. Si se tiene en cuenta el soporte a la depuración y enfoque de diseño.

3.3.2.5 Tiempo empleado en el análisis sintáctico.

Cuanto tiempo consideras que te ha llevado la implementación del sintáctico?

13 respuestas

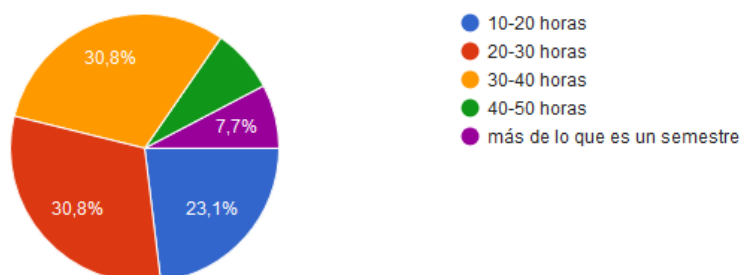


Ilustración 7 – Diagrama Tiempo empleado en el análisis sintáctico

- Conclusión:

Sin tener en cuenta el último comentario podemos concluir que la realización del análisis léxico lleva menos de 40 horas para la gran mayoría de los alumnos, por lo que se considera que esta parte requiere un estudio más exhaustivo que el análisis léxico, pudiendo beneficiar en más tiempo a los alumnos.

3.3.2.6 Dificultades y carencias del análisis sintáctico

Que dificultades has tenido y que has echado en falta?

7 respuestas

Ejemplos resueltos
Algunas ambigüedades.
Es una parte un poco mas larga que el léxico, pero en cuanto se entiende lo que hay que hacer es muy mecanica.
Algo que localice mejor los errores dentro del codigo.
ejemplos, documentacion
Detección de fallos
Ninguna, una visión del árbol completo

Ilustración 8 – Comentarios Dificultades y carencias análisis sintáctico

- Conclusión:

De nuevo, no es objetivo de este proyecto generar documentación auxiliar, por lo que nos centramos en los comentarios más útiles sobre la detección de fallos y visión del árbol completo.

3 - Especificación del proyecto

3.3.2.7 Tiempo empleado en el análisis semántico.

Cuanto tiempo consideras que te ha llevado la implementación del semántico?

13 respuestas

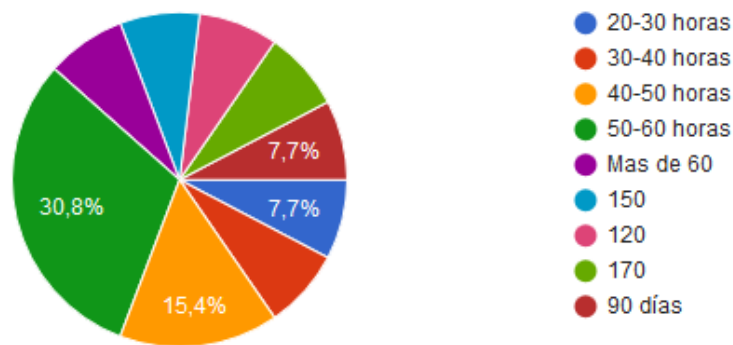


Ilustración 9 – Diagrama Tiempo empleado en el análisis semántico

- Conclusión:

Podemos observar datos bastante dispares, desde 20 horas hasta 170. Con los datos obtenidos queda claro que el análisis semántico es bastante complejo, pues el 85% de los alumnos encuestados han tardado más de 40 horas en su realización. Cabe destacar los casos de más de 100 horas, lo que indica una mala planificación, o dificultades en la comprensión de las tareas a realizar.

En esta parte se prestará especial atención a como aligerar el trabajo de los alumnos y ayudarles a comprender las tareas a realizar, así como estructurar mejor el código lo que beneficiará su comprensión.

3.3.2.8 Dificultades y carencias del análisis semántico

Que dificultades has tenido y que has echado en falta?

9 respuestas

No es muy dificultoso, solo trabajoso.
Tuve que rehacer todo el semantico. Pocos ejemplos y realmente mucha dificultad para entender que es lo que hay que hacer
Muchas dificultades para saber como empezar, se echa de menos un ejemplo por parte del ED para poder comenzar
Ser más específico con los requisitos mínimos. Todas las verificaciones son muchísimas. Es posible que dediqué más tiempo del estrictamente necesario
Lo mismo que en el sintactico y explicaciones iniciales por parte del ED.
falta de test, buena documentacion, ejemplos
No saber cuál era el objetivo. Mucho tiempo perdido sin hacer nada.
Ninguna, resaltado de sintaxis en cup
soporte a la depuración

Ilustración 10 – Comentarios Dificultades y carencias análisis semántico

- Conclusión:

En esta pregunta parece que el problema clave se encuentra en definir bien el alcance del análisis sintáctico. Esto no es objetivo del proyecto, no obstante, se tiene en cuenta el soporte a la depuración y el resaltado de sintaxis en cup.

3.3.2.9 Tiempo empleado en el código intermedio.

Cuanto tiempo consideras que te ha llevado la implementación del código intermedio?

13 respuestas

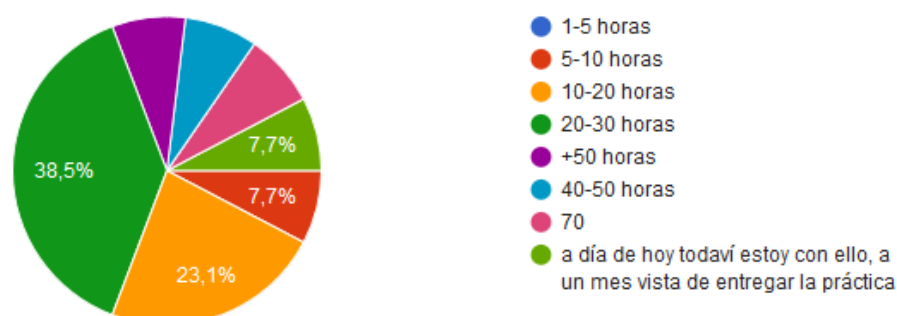


Ilustración 11 – Diagrama Tiempo empleado en el código intermedio

3 - Especificación del proyecto

- Conclusión:

En esta pregunta podemos concluir que aproximadamente el 75% de los alumnos realiza el código intermedio en menos de 30 horas, por lo que esta parte no será de demasiado interés dado que no va a producir un ahorro importante de tiempo.

3.3.2.10 Dificultades y carencias del código intermedio.

Que dificultades has tenido y que has echado en falta?

7 respuestas

Ejemplos practicos como los de gamboa (en las transparencias es todo muy genérico)
No sabes lo que estas haciendo hasta que llegas al final. Luego tienes que tocar muchas cosas.
Sencillo
Lo mismo que en el semantico.
falta de documenacion, ejemplos claros
Información sobre que tenía que hacer
Ninguna, nada

Ilustración 12 – Comentarios Dificultades y carencias código intermedio

- Conclusión:

La carencia en esta parte es la falta de documentación, lo que no es objetivo del proyecto.

3.3.2.11 Tiempo empleado en la generación de código final.

Cuanto tiempo consideras que te ha llevado la implementación del código final?

13 respuestas

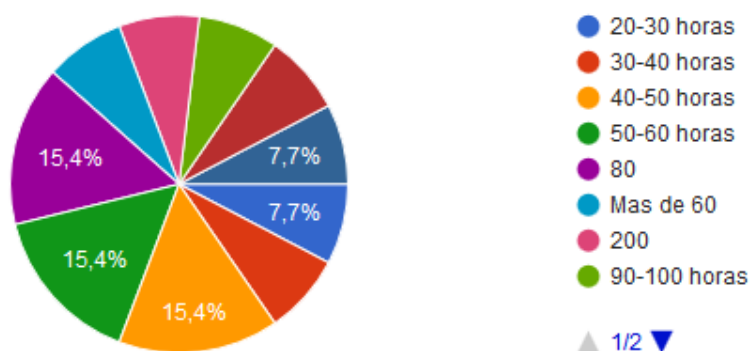


Ilustración 13 – Diagrama Tiempo empleado en el código final

- Conclusión:

Como es de esperar por los datos de la primera pregunta, puede observarse que esta parte es compleja viendo los datos obtenidos. La media de tiempo requerido está en torno a las 60 horas, pero con resultados tan dispares es de esperar que aquí mucha gente encuentre problemas.

Siendo uno de los resultados “20-30 horas”, es de suponer que estos problemas seguramente en gran parte sean debidos a la falta de información y documentación, sin embargo, habrá que buscar una forma de reducir los tiempos altos, ya sea estructurando el código o facilitando algún tipo de ayuda en el direccionamiento de las variables del lenguaje.

3.3.2.12 Dificultades y carencias del código final.

Que dificultades has tenido y que has echado en falta?

9 respuestas

Comprender el RA, los tipos de direccionamientos, y ejemplos resueltos
Muchas, la documentación sobre como se implementa el código final es penosa, realmente no tenía ni idea de como implementar el RA y cuando he llegado a esta fase he tenido que cambiar muchas cosas del parser y me han hecho perder mucho tiempo. Falta teoría buena y no el libro que hay.
La parte mas difícil de la ped sin duda, realmente no hay por donde pillarlo.
el depurador para nullpointer
Explicaciones y ejemplos, un video hubiese sido muy muy valorado. Sobre todo en la parte de funciones y procedimientos.
falta de buena documentacion, ejemplos
Problemas finales en cuanto a invocación de funciones. No poder detectar los fallos por línea.
Ninguna, más capacidad de ens2001
la curva de aprendizaje es muy alta... SEMANTICO+CI+CF > un semestre

Ilustración 14 – Comentarios Dificultades y carencias código final

- Conclusión:

Queda claro que lo que requieren los alumnos de nuevo es documentación y ejemplos. El mayor problema encontrado parece ser el direccionamiento de los Registros de Activación (al realizar llamadas a funciones del lenguaje). Se tendrá en cuenta el comentario de la detección de los fallos por línea. También se prestará especial atención

3 - Especificación del proyecto

a los errores “NullPointerException” que se producen a veces en la ejecución y cuya traza no ayuda a localizar el error.

En cuanto a ens2001, la herramienta que simula un intérprete de código ensamblador, no se considera que necesite cambios.

3.3.2.13 Mejoras en las prácticas

Que mejoraríais en general en cuanto a las peds de PL1 y PL2

9 respuestas

Ejemplos resueltos
Mayor documentación que la que hay.
Estaría bien poder ver donde se producen los errores
La práctica debería empezar el TLP (léxico y sintactico) - PI1 (semántico) - PL2 (CI y código final)
Pondría videos del equipo docente explicando como se hacen partes de la práctica.
mejorar la gestion de errores por parte del IDE, proponer mas videos de resolucion de practicas pasadas
Que te enseñen el objetivo a lograr y como llegar a ello mediante vídeos. La cantidad de tiempo empleada es ingente y no te permite luego estudiar el examen. Es muy desesperante.
Una reescritura de las transparencias disponibles
faltan ejemplos para cada módulo concreto hasta el final, con SEMANTICO+CI+CF , tener una visión global de qué hace y debe hacer cada cosa...

Ilustración 15 – Comentarios Mejoras en las peds

- Conclusión:

La elaboración de documentación y videos es lo que más requieren los alumnos. Debido a que no es objetivo del proyecto, nos centramos solo en los dos comentarios que requieren depuración de errores.

3.3.2.14 Enunciados en las prácticas

Que os han parecido los enunciados proporcionados?

9 respuestas

Muy completos hasta la parte semántica, de ahí en adelante muy escasos
Los enunciados no están mal, yo creo que el mayor problema es que no hay ningún ejemplo de lo que tienes que hacer y te sientes muy perdido y solo, muchas veces si no fuera por el foro de no sabrías por donde continuar y no debería ser así, porque al final cuando la hacer ves como realmente debería ser y que quizás con algo más de información no te sentirías como un niño que ha perdido a sus padres en medio de un centro comercial.
Muy completos
NO, PL2 es mucho más costosa. PL1 está dentro de un baremo razonable
Los enunciados estan bien.
explicativos pero no aclaratorios
Con errores
Sí, quizá un poco simple el lenguaje.
faltan ejemplos más precisos de cada módulo

Ilustración 16 – Comentarios Opinión de los enunciados

- Conclusión:

Los alumnos coinciden en que los enunciados son lo bastante completos, a falta de aclarar el alcance de cada parte y más ejemplos sobre todo en la parte del código final e intermedio.

3.3.2.15 Errores y localización

En cuanto a los errores y como localizarlos?

6 respuestas

Necesitamos una herramienta para detectar los errores Null Pointer Exception a la hora del desarrollo del código final.
Aun tengo pesadillas.
por consola, muy lentamente
Horribles
Que flex y cup lancen información un poco más ordenada
falta el soporte a la depuración del código CUP, no es intuitivo cuando falla.

Ilustración 17 – Comentarios Opinión sobre errores

3 - Especificación del proyecto

- Conclusión:

Este parece ser el punto en el que más sufren los estudiantes definiendo la localización de errores como una “pesadilla”. Se estudiará a fondo el tema.

3.3.2.16 Otros comentarios

Algún otro comentario sobre las peds?

6 respuestas

Lo mejor de estas ped es aprobarlas jajaja
Mi documentación y pruebas ha sido muy extensa. Equivalente a un PFG
Al final da gusto haberla acabado, porque se aprende mucho. Pero con un poco de ayuda se aprendería lo mismo y gustaría más.
Proponer mas videos de implementacion

Ilustración 18 – Comentarios Otros comentarios

- A partir de este apartado no hay conclusiones que sacar, simplemente los alumnos requieren más ayuda, y este es el propósito de este proyecto de fin de grado.

3.3.3 Especificación del alcance de la herramienta

La herramienta debe ser útil para el desarrollo del compilador sin menospreciar los conocimientos que debe adquirir el alumno en las asignaturas. Se descartan, por tanto, ideas como generación automática de código o tratar de incluir partes comunes para que el alumno no tenga que hacerlas. La finalidad y el alcance debe ser ahorrar tiempo escribiendo código y facilitar la depuración, dado que está última es bastante ambigua en CUP.

3.3.4 Diseño de la herramienta

Una vez identificados los puntos clave se propone como solución:

- Dejar visible para los alumnos el código original para que puedan comprobar los cambios que se producen con el uso de la herramienta.
- Desarrollar herramientas independientes para cada una de las asignaturas. Cada una contará con su propia parte de depuración de código.
- En cada herramienta habrá partes independientes sobre las distintas fases del compilador, facilitando así la distinción entre las tareas que se deben realizar a cada paso.

3.3.5 Temporización

La planificación se realiza considerando que el proyecto debe englobarse dentro de los 18 créditos de la asignatura PFG y contar con una duración de 450h.

El proyecto se divide en dos grandes fases, una por cada herramienta que se va a desarrollar. En la imagen 3 se propone el diagrama de Gantt correspondiente con la planificación, siendo cada casilla de aproximadamente 20 horas dedicadas.

Actividades	Febrero				Marzo				Abril				Mayo				Junio			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1 Investigación preliminar																				
2 Especificacion del alcance																				
3 Diseño de la herramienta																				
4 Implementacion																				
5 Pruebas																				
6 Documentacion																				

Ilustración 19 – Temporización del proyecto

4 Análisis del sistema

4 - Análisis del sistema

Este capítulo explica en que consiste el proyecto y cuál es su funcionamiento. Para ello se explicarán los requisitos del sistema y los diferentes casos de uso.

4.1 Descripción del sistema

El proyecto consiste en el diseño e implementación de herramientas de apoyo a la realización de las distintas fases de un compilador.

A partir de los datos obtenidos de la encuesta y la experiencia propia en el desarrollo del compilador se propone que el proyecto conste de los siguientes apartados:

- **Procesadores de Lenguajes 1:**
 - Scanner.flex: Editor de texto plano que alberga el código original perteneciente a la parte de análisis léxico.
 - Parser.cup: Editor de texto plano que alberga el código original perteneciente a la parte de análisis sintáctico.
 - Léxico: Separación de las estructuras propias de JFLEX mediante formularios y tablas para el ingreso de tokens, expresiones, errores y símbolos de comentarios.
 - Sintáctico: Separación de las estructuras propias de CUP mediante la separación de los “no terminales” y “tokens” del lenguaje.
 - Tests: Orientado a la depuración de código y creación del árbol sintáctico.
- **Procesadores de Lenguajes 2:**
 - Parser.cup: Editor de texto plano que alberga el código original perteneciente a la parte de análisis sintáctico.
 - Semántico: Separación de las estructuras propias de CUP mediante la separación de los distintos “no terminales” en pestañas en las que se puede encontrar su código semántico.
 - Código final: Separación de las distintas partes del código final para facilitar su traducción independiente.
 - Tests: Depuración del código mostrando contenido del test, ejecución paso a paso, y actualización de las tablas de símbolos y tipos.

4.1.1 Características de los usuarios

El propósito principal de este proyecto es facilitar el trabajo de los alumnos de la UNED, no obstante, puede ser usado por otros teniendo en cuenta que es una herramienta educativa y no profesional ni comercial.

4.1.2 Funcionalidades

4.1.2.1 Léxico:

- a. Añadir/Eliminar tokens
- b. Añadir/Eliminar expresiones
- c. Añadir/Eliminar errores
- d. Añadir símbolos de comentarios
- e. Generar el código correspondiente

4.1.2.2 Sintáctico

- a. Añadir no terminal
- b. Eliminar no terminal
- c. Guardar no terminal
- d. Editar no terminal
- e. Añadir producción
- f. Eliminar producción
- g. Añadir componentes a la producción
- h. Guardar el código correspondiente

4.1.2.3 Tests Procesadores de Lenguajes 1

- a. Cargar SubParser
- b. Probar Test
- c. Exportar árbol sintáctico a fichero

4.1.2.4 Semántico y Código intermedio

- a. Aumentar tamaño del texto
- b. Disminuir tamaño del texto
- c. Introducción de código mediante botones

4 - Análisis del sistema

4.1.2.5 Código final

- a. Aumentar tamaño del texto
- b. Disminuir tamaño del texto
- c. Capturar código intermedio
- d. Generar DATA (Asignación de espacio para cadenas)
- e. Generar MEMORY (Asignación de direcciones de memoria)

4.1.2.6 Tests Procesadores de Lenguajes 2

- a. Cargar Test
- b. FIN (Recorre toda la ejecución)
- c. Ejecución paso a paso ">"
- d. Muestreo del contenido del test
- e. Muestreo del contenido de la tabla de símbolos
- f. Muestreo del contenido de la tabla de tipos
- g. Muestreo de la ejecución paso a paso

4.1.3 Restricciones generales

La principal restricción del proyecto es el objetivo de evitar la necesidad de invertir presupuesto en el proyecto debido a su finalidad educativa. Se pretende realizar una herramienta gratuita, de código libre.

4.2 Casos de uso

En este apartado se describen los casos de uso proporcionados por cada una de las aplicaciones desarrolladas.

4.2.1 Herramienta de elaboración de compilador

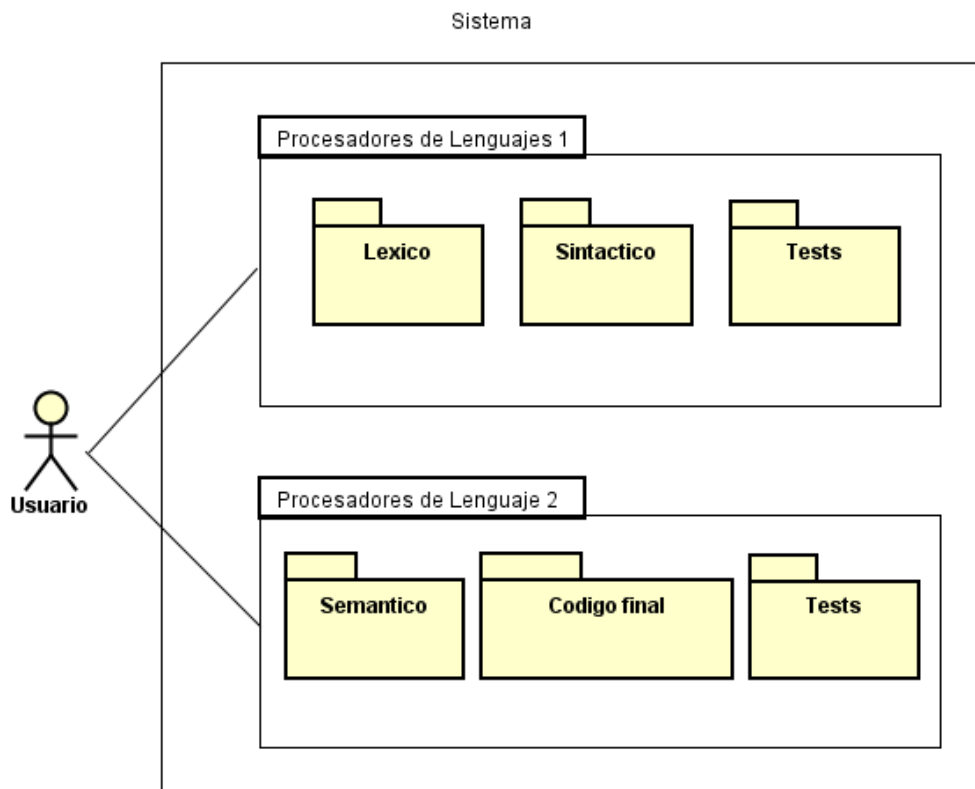


Ilustración 20 – Caso de uso 0 – Sistema completo

El proyecto completo consiste en 2 aplicaciones (plugins) independientes que a su vez tienen 3 componentes diferenciados cada uno. Los paquetes de este caso de uso agrupan todos aquellos casos de uso pertenecientes a los componentes del proyecto.

Cabe destacar que el uso de cualquiera de los componentes no depende de que anteriormente se haya usado la herramienta para generar el resto, por lo que cualquier parte puede funcionar por sí sola.

Cada herramienta posee su propio componente “Tests” debido a que las comprobaciones que se realizan son totalmente diferentes.

4 - Análisis del sistema

4.2.2 Análisis Léxico

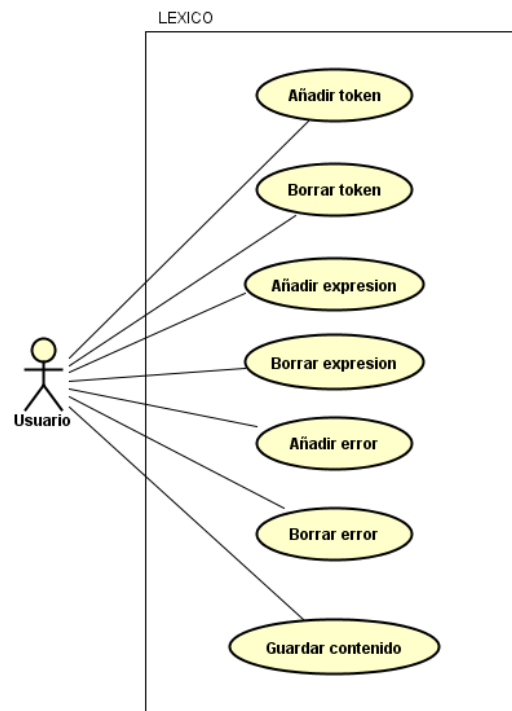


Ilustración 21 – Caso de uso 1 – Léxico

Casos de uso:

- Añadir token: Es la operación que permite añadir un token al lenguaje en construcción.
- Borrar token: Es la operación que permite eliminar un token del lenguaje en construcción.
- Añadir expresión: Es la operación que permite añadir expresiones (patrones) al lenguaje en construcción.
- Borrar expresión: Es la operación que permite eliminar una expresión del lenguaje en construcción.
- Añadir error: Es la operación que permite añadir expresiones de errores en el lenguaje en construcción.
- Borrar error: Es la operación que permite eliminar una expresión de error del lenguaje en construcción.
- Guardar contenido: Es la operación que permite guardar los cambios realizados en el fichero destino (scanner.flex).

4.2.3 Análisis sintáctico

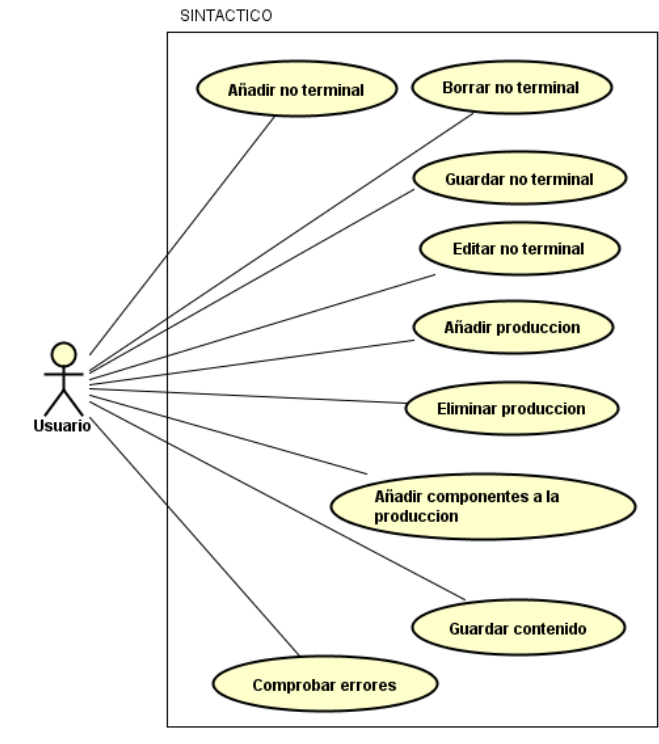


Ilustración 22 – Caso de uso 2 - Sintáctico

Casos de uso:

- Añadir no terminal: Permite añadir no terminales al lenguaje en construcción.
- Borrar no terminal: Permite eliminar no terminales al lenguaje en construcción.
- Guardar no terminal: Es la operación que permite guardar un no terminal.
- Editar no terminal: Es la operación que permite editar el contenido de un no terminal ya creado con anterioridad.
- Añadir producción: Es la operación que permite añadir una nueva producción a un no terminal ya existente.
- Eliminar producción: Es la operación que permite eliminar una producción de un no terminal ya existente.
- Añadir componentes a la producción: Es la operación que permite añadir tokens o no terminales a una producción.
- Guardar contenido: Es la operación que permite guardar los cambios realizados en el fichero destino.
- Comprobar errores: comprueba si existen errores en los no terminales.

4 - Análisis del sistema

4.2.4 Tests PL1

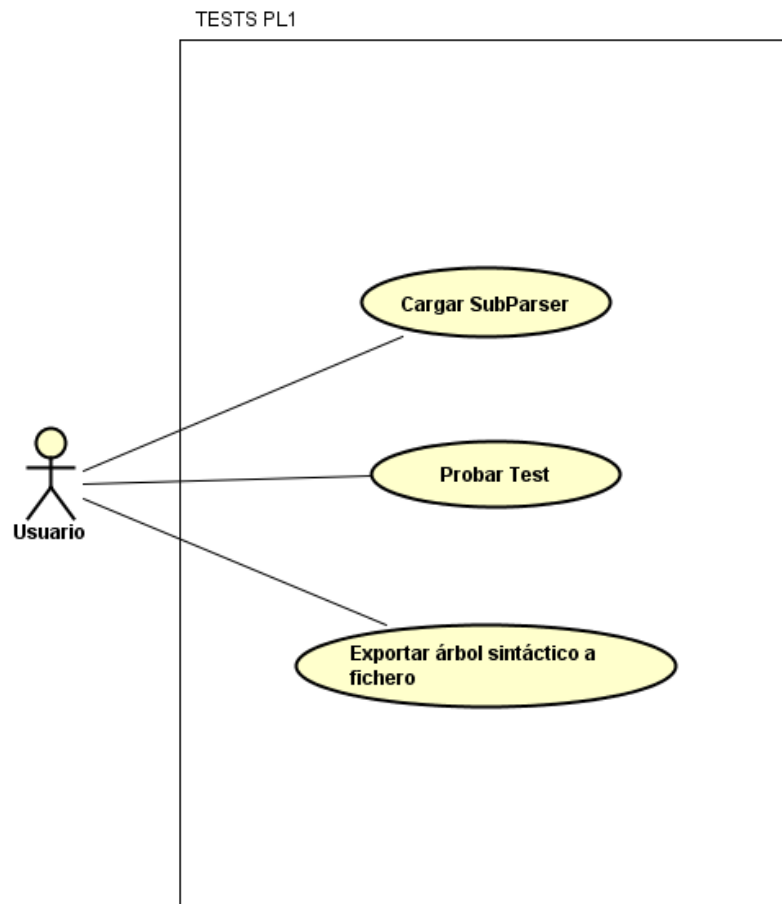


Ilustración 23 – Caso de uso 3 – Tests PL1

Casos de uso:

- Cargas SubParser: Esta operación genera la clase java encargada de realizar los test del lenguaje.
- Probar test: Esta operación que ejecuta las comprobaciones sobre un ejemplo de código de entrada y produce un árbol sintáctico si es correcto.
- Exportar árbol sintáctico a fichero: Esta operación guarda el árbol sintáctico anteriormente generado en un fichero de texto.

4.2.5 Análisis Semántico y código intermedio

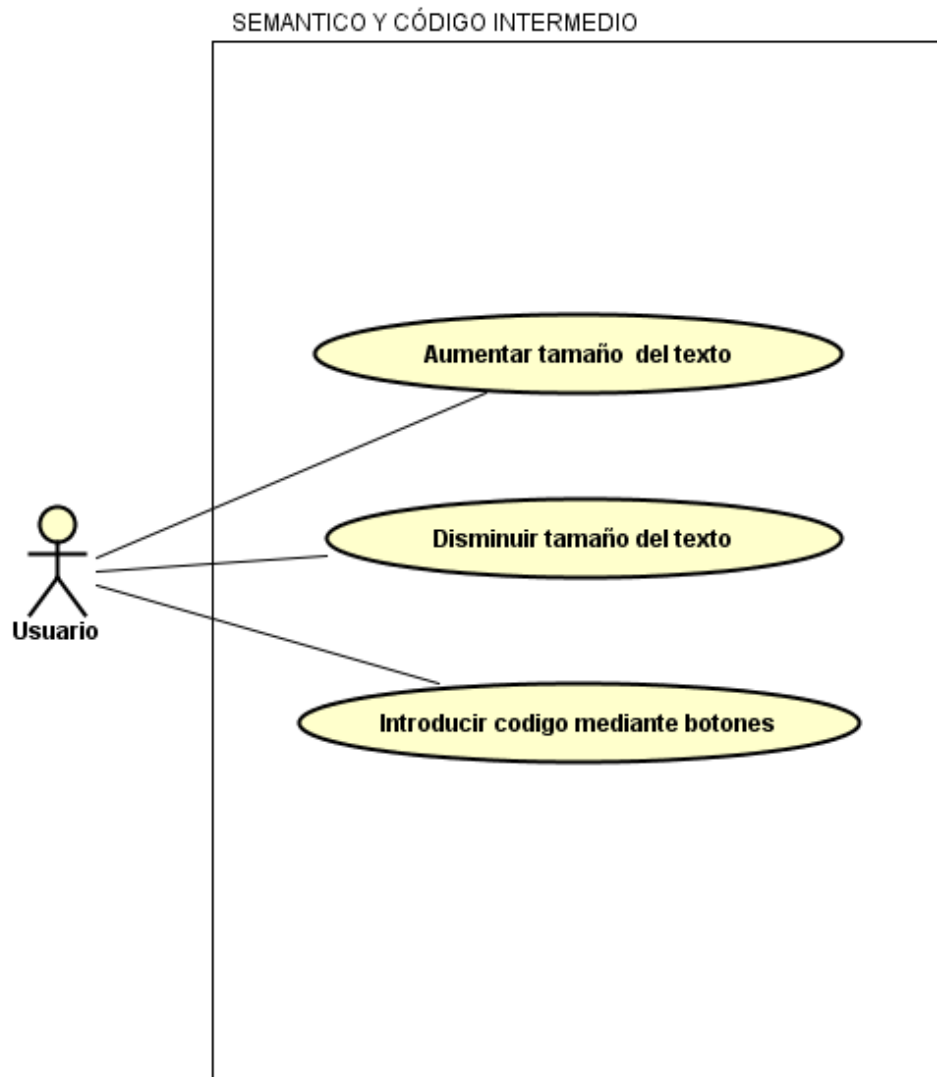


Ilustración 24 – Caso de uso 4 – Semántico y código intermedio

Casos de uso:

- Aumentar tamaño del texto: Es la operación que aumenta la fuente del texto mostrado por pantalla.
- Disminuir tamaño del texto: Es la operación que disminuye la fuente del texto mostrado por pantalla.
- Introducir código mediante botones: Esta operación consiste en arrastrar un botón al espacio de edición de texto, y añade un texto predefinido correspondiente con el contenido del botón.

4 - Análisis del sistema

4.2.6 Código final

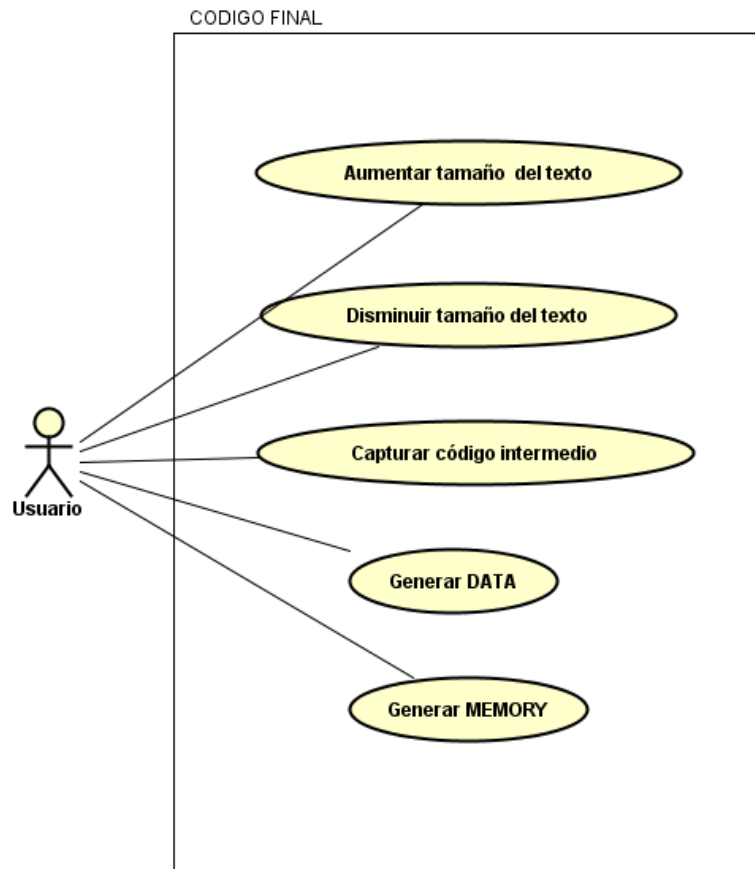


Ilustración 25 – Caso de uso 5 – Código final

Casos de uso:

- Aumentar tamaño del texto: Es la operación que aumenta la fuente del texto mostrado por pantalla.
- Disminuir tamaño del texto: Es la operación que disminuye la fuente del texto mostrado por pantalla.
- Capturar código intermedio: Es la operación que extrae los nombres de las operaciones de código intermedio declaradas.
- Generar DATA: Esta operación genera una clase java encargada del tratamiento de cadenas de texto del lenguaje en construcción.
- Generar MEMORY: Esta operación genera una clase java encargada de asignar posiciones de memoria a los distintos símbolos presentes en la tabla de símbolos del lenguaje en construcción.

4.2.7 Tests PL2

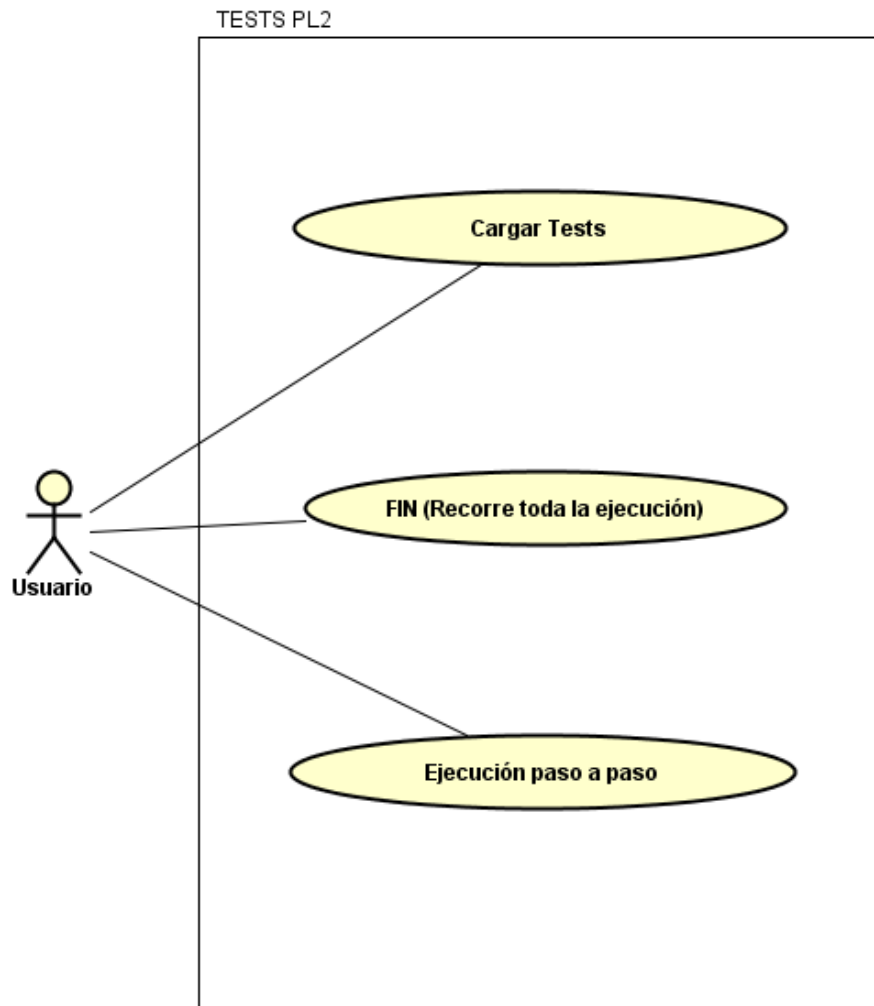


Ilustración 26 – Caso de uso 6 – Tests PL2

Casos de uso:

- Cargar Tests: Es la operación encargada de cargar un test seleccionado en memoria y preparar su ejecución.
- FIN: Es la operación encargada de ejecutar el código del test seleccionado completo.
- Ejecución paso a paso: Esta operación avanza un paso en la ejecución del test seleccionado.

5 Diseño e implementación

5 - Diseño e implementación

En este capítulo se aborda el tema del diseño y la implementación del proyecto y cada una de sus partes.

5.1 Estudio y selección de tecnologías

En cuanto a los requisitos para ejecutar la aplicación, cualquier sistema en el que pueda ejecutarse Eclipse y Java, en concreto JRE o JDK 8 será compatible. Se detallan los requisitos del sistema para Java 8:

- RAM: 128 MB
- Espacio en disco: 124 MB
- Procesador: Mínimo Pentium 2 a 266 MHz

A los requisitos de espacio en disco habría que añadir 5 MB para la herramienta y el tamaño de la arquitectura del Equipo Docente lo que hace un total de 130 MB. El sistema debe tener instalado Eclipse 3.4 o superior.

5.1.1 JFlex³

JFlex es un generador de analizadores léxicos para Java, escrito en Java. Un generador de analizadores léxicos toma como entrada una especificación con un conjunto de expresiones regulares y acciones correspondientes. Genera un programa que lee la entrada, compara la entrada con las expresiones regulares en el archivo de especificación, y ejecuta la acción correspondiente si coincide con expresión regular. Este suele ser el primer paso de front-end en los compiladores, haciendo coincidir palabras clave, comentarios, operadores, etc., y generando un flujo de tokens de entrada para los analizadores.

Los lectores JFlex se basan en autómatas finitos deterministas (DFA). Son rápidos, sin costosos retrocesos.

JFlex está diseñado para trabajar junto con el generador de analizadores LALR CUP de Scott Hudson, y la modificación Java de Berkeley Yacc BYacc / J de Bob Jamison.

³JFlex The Fast Scanner Generator for Java [en línea], <http://www.jflex.de/> [Consulta 20/02/18]

5.1.2 CUP (Construction of Useful Parsers)⁴

En la realización del compilador en las asignaturas de Procesadores de Lenguajes se utiliza CUP como generador de analizadores LALR. Fue desarrollado por C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel y Michael Petter. Implementa la generación de un analizador LALR(1) estándar.

Se ha investigado el código fuente de CUP para poder realizar partes del proyecto debido a que la documentación no abarca todas las posibilidades del generador ni su funcionamiento interno a la hora de crear el analizador LALR.

5.1.3 Window Builder⁵

WindowBuilder es un plugin de eclipse compuesto por SWT Designer y Swing Designer y hace que sea muy fácil crear aplicaciones Java GUI sin perder mucho tiempo escribiendo código. Con el uso del diseñador visual WYWSIWYG y las herramientas de diseño se pueden crear formularios simples para ventanas complejas, generando el código Java correspondiente.

El complemento crea un árbol de sintaxis abstracta (AST) para navegar por el código fuente y utiliza GEF para visualizar y administrar la presentación visual.

El código generado no requiere ninguna biblioteca personalizada adicional para compilar y ejecutar: todo el código generado se puede utilizar sin tener instalado WindowsBuilder Pro.

5.1.4 Sistema operativo

LINUX: es un sistema operativo gratuito, altamente reconocido por su seguridad y estabilidad. Además, está sufriendo una notable expansión debido al reconocimiento por parte de Microsoft y su inclusión en sus sistemas.

⁴ CUP, LALR Parser Generator for Java [en línea], <http://www2.cs.tum.edu/projects/cup/> [Consulta 20/02/18]

⁵ WindowBuilder | The Eclipse Foundation [en línea], <https://www.eclipse.org/windowbuilder/> [Consulta 20/02/18]

5 - Diseño e implementación

WINDOWS: es un sistema operativo comercial, el más conocido y distribuido en los ordenadores personales de todo el mundo.

Dado que Java es multiplataforma, la aplicación puede ejecutarse en cualquier sistema operativo que soporte eclipse, ya que son plugins preparados para funcionar con eclipse. Es una herramienta de desarrollo de un compilador, por lo que su uso está destinado específicamente para ordenadores, no teniendo en cuenta otro tipo de dispositivos (tablets, móviles). Se espera que su uso se haga principalmente en Windows o Linux. El desarrollo de la aplicación se ha realizado en un equipo con sistema operativo Windows 10.

5.1.5 Lenguaje de programación

Se ha elegido Java⁶ debido a su facilidad de uso y su compatibilidad por completo con las herramientas utilizadas en las asignaturas de Procesadores de Lenguajes (JFlex, CUP).

Java es un lenguaje de programación orientado a objetos y multiplataforma. Es un lenguaje asentado con mucha documentación útil para el desarrollo de aplicaciones. Debido al gran uso del lenguaje es fácil encontrar distintos plugins o funcionalidades que ayuden al desarrollo de componentes. Se mencionan, a continuación, las dos más importantes utilizadas en este proyecto.

- **Reflexión⁷**

Una de las funcionalidades más potentes y poco conocidas de Java es su soporte para la reflexión. Java Reflection es quizás la API que más versatilidad aporta al lenguaje Java ya que nos permite resolver muchos problemas de una forma totalmente diferente a la habitual. Mediante su uso el programador puede inspeccionar y manipular clases e interfaces (así como métodos y campos) en tiempo de ejecución, sin conocer a priori (en tiempo de compilación) los tipos y/o nombres de las clases específicas con las que está trabajando. En un principio no es nada sencillo de entender, pues la idea de un programa dinámico que pueda modificarse una vez ejecutado no se enseña en la universidad.

⁶ Oracle, Software Java [en línea], <https://www.oracle.com/es/java/index.html> [Consulta 20/02/18]

⁷Oracle, The Reflection API [en línea], <https://docs.oracle.com/javase/tutorial/reflect/> [Consulta 20/02/18]

Con la necesidad de modificar clases del proyecto del alumno surge el problema de la creación dinámica de clases y su compilación y ejecución de forma dinámica estando el plugin ya ejecutado. Dichas clases nuevas además deben trabajar junto con las ya existentes del alumno, así como con la librería CUP.

Para conseguir los objetivos del proyecto y poder capturar la traza de la ejecución es necesario modificar la clase `parser.java` y ejecutar `subparser.java` que se ocupa de capturar todos los mensajes generados, así como errores. No se ha propuesto capturar la línea exacta del error por no coincidir `parser.cup` con `parser.java`, puesto que el segundo fichero se elabora a partir del primero, y es el segundo el que se ejecuta.

- **Patrones⁸**

Un patrón es una expresión regular que representa un conjunto de caracteres. Básicamente un patrón puede representar infinidad de cadenas que cumplan con la expresión regular que describe. Se han utilizado patrones en la lectura de los ficheros de entrada `scanner.flex` y `parser.cup` debido a la estructura de los ficheros.

Un ejemplo de patrón es `[0-9]` que coincide con cualquier número de 1 sola cifra comprendido entre el 0 y el 9.

En el presente proyecto se han utilizado patrones para poder separar los distintos tokens en `scanner.flex` y los no terminales y sus producciones en `parser.cup`.

⁸ Java Platform SE 7, Pattern [en línea],
<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>
[Consulta 23/02/18]

5 - Diseño e implementación

5.2 Diagramas de clases

En este apartado se representarán los diferentes diagramas de clases del proyecto.

5.2.1 Diagrama de Clases Procesadores de Lenguajes 1



Ilustración 27 – Diagrama de clases – Paquetes PL1

Se muestran los principales paquetes del proyecto de procesadores de lenguajes 1 y se detallan a continuación.

5.2.1.1 Plugeditor.editors

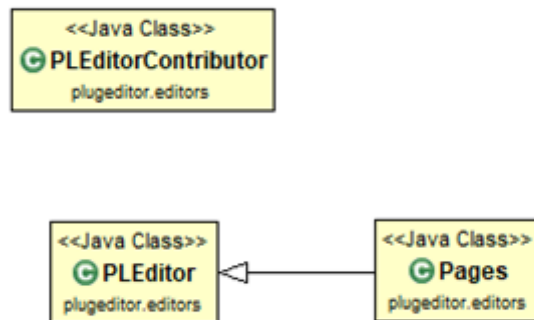


Ilustración 28 – Diagrama de clases – PlugEditor

Este paquete contiene la estructura principal de la aplicación. Se ocupa del tratamiento de las diferentes pestañas del plugin, así como el tipo de cada una de ellas, acciones a realizar al cambiar entre pestañas, diseño del logo y la ejecución.

- PLEditorContributor: es la clase encargada de capturar la acción que realiza el cambio entre pestañas y realizarlo.
- PLEditor: es la clase principal de este paquete. Se encarga de definir las acciones que han de ser realizadas en cada página (pestaña) del plugin. Contiene un conjunto de páginas, inicializa el plugin y mantiene funciones para el acceso y modificación de clases del proyecto.
- Pages: Constructor de páginas (pestañas).

5.2.1.2 Léxico

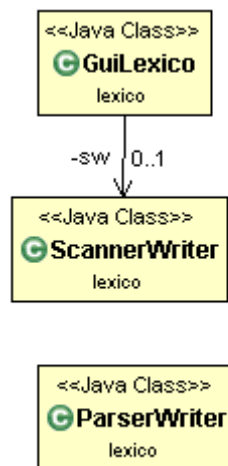


Ilustración 29 – Diagrama de clases – Léxico

Este paquete contiene el tratamiento de tokens correspondiente al análisis léxico del compilador. Se ocupa de la generación de los ficheros `scanner.flex` y `parser.cup` del proyecto destino en lo que se refiere a la parte de generación de tokens.

- **GuiLexico:** Esta clase es la que mantiene la interfaz gráfica de la pestaña “léxico”. Mantiene listas de tokens, expresiones, errores y comentarios, además de tablas con su contenido y el formulario para rellenarlas. Realiza la carga de las listas a partir del fichero `scanner.flex`.
- **ScannerWriter:** Es la clase encargada de guardar el contenido de las listas en `scanner.flex` una vez que estén rellenas y se pulse el botón de guardado.
- **ParserWriter:** Es la clase encargada de declarar los tokens en la clase `parser.cup` (necesario para el correcto funcionamiento del compilador)

5 - Diseño e implementación

5.2.1.3 Common

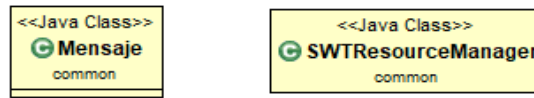


Ilustración 30 – Diagrama de clases – Common

Este paquete contiene clases útiles para todo el proyecto.

- Mensaje: Es la clase encargada de mostrar mensajes por pantalla de forma gráfica, ya sean avisos o errores.
- SWTResourceManager: Es la clase que se encarga del tratamiento de imágenes o texto, permitiendo ampliar/disminuir la fuente o aplicarle estilos a la fuente como negrita, subrayado.

5.2.1.4 Sintáctico

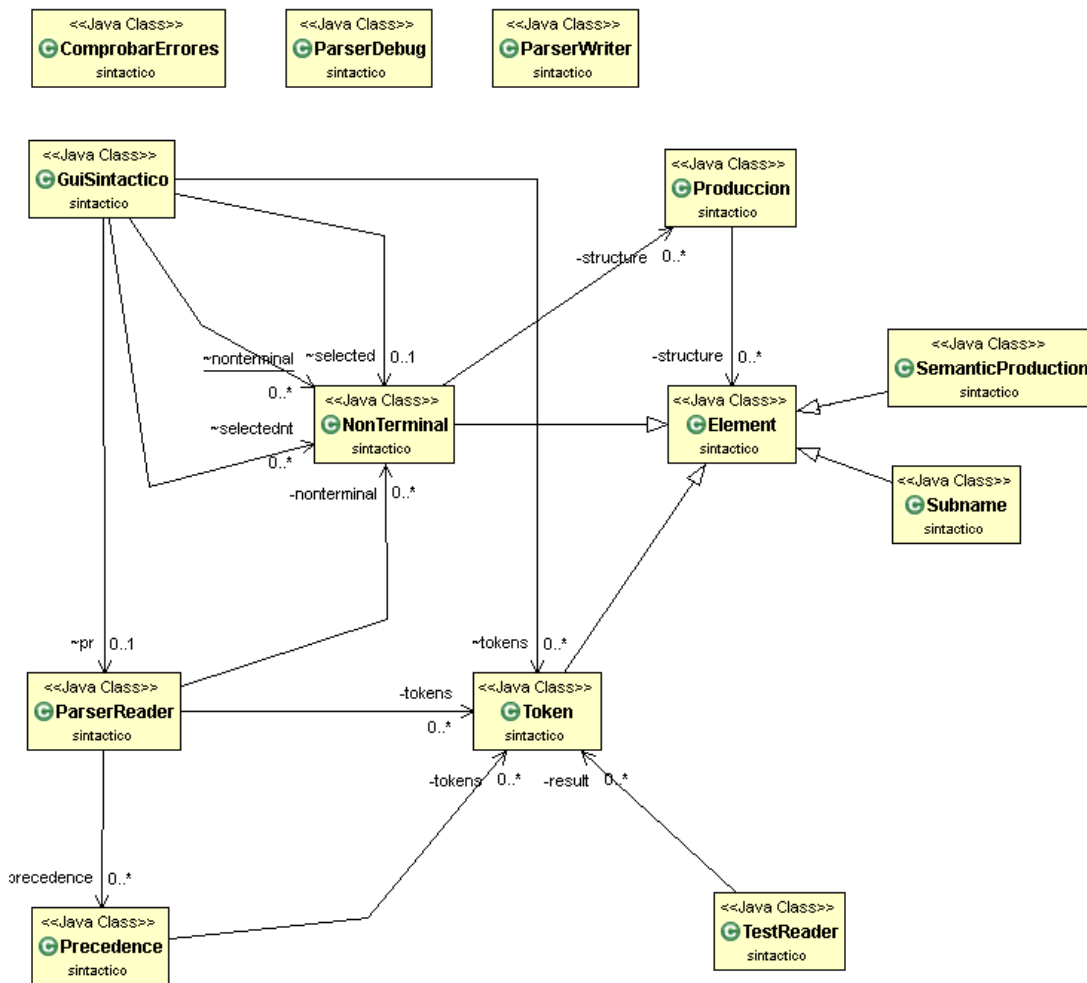


Ilustración 31 – Diagrama de clases – Sintáctico

Este paquete se encarga del tratamiento de no terminales, su inclusión, eliminación y modificación. Se realiza la comprobación de errores y la lectura/escritura en el fichero parser.cup.

Para entender el diagrama es necesario aclarar los siguientes conceptos:

Un símbolo terminal (token) es una de las cadenas del lenguaje.

Un símbolo no terminal actúa a modo de variable. Cada símbolo representa un conjunto de cadenas que se derivan a partir de ese símbolo.

Una producción representa la definición recursiva de un lenguaje.

5 - Diseño e implementación

Así pues, consideramos que tanto token como Nonterminal extienden de la clase Element. Cada NonTerminal puede tener varias producciones, cada una de las cuales puede estar compuesta por 0 o más elementos.

Cabe destacar que se tiene en cuenta a la hora de la lectura del fichero parser.cup la posibilidad de que haya producciones semánticas o nombres de los distintos elementos. Éstos se ignoran en la generación de la gramática, pero si existen en la carga del fichero se guardan para que puedan ser escritos de nuevo en el fichero cuando se finalice el trabajo.

Se propone el siguiente ejemplo de gramática sencilla para comprender los conceptos mencionados:

```
Acción -> Animal:an { : System.out.print(an); :} Sonido Lugar;  
Animal -> perro | loro;  
Sonido -> ladra;  
Sonido -> canta;  
Lugar -> en casa;
```

Se detallan los componentes de la gramática:

Terminales (tokens): perro, loro, ladra, canta, en casa

No terminales: Acción, Animal, Sonido, Lugar

Producción de Acción: "Acción -> Animal Sonido Lugar"

Producciones de Animal: "Animal -> perro", "Animal -> loro"

Producciones de Sonido: "Sonido-> ladra", "Sonido -> canta"

Producción de Lugar: "Lugar -> en casa"

Nombre del elemento Animal: "an"

Producción semántica: La que está encerrada entre "{:" y ":", en este caso "System.out.print(an);" que además hace referencia al elemento "an".

Habiendo aclarado cuales son los componentes de una gramática detallamos las clases que componen el paquete:

- **ComprobarErrores:** Clase encargada de comprobar los posibles errores en la gramática. Se centra en la comprobación de duplicados entre los no terminales y la comprobación de que todos los no terminales utilizados en las producciones hayan sido declarados previamente.
- **ParserDebug:** Clase encargada de leer la traza de ejecución de los tests y filtrar la información relevante.
- **ParserWriter:** Clase encargada de la escritura del fichero parser.cup a partir de una lista de no terminales.
- **Element:** Clase padre para los elementos de la gramática, contiene el nombre del elemento, así como los métodos para comprobar si un elemento es igual a otro.
- **Token:** Clase encargada de la construcción de tokens.
- **NonTerminal:** Clase encargada de la construcción de no terminales. Mantiene una lista de producciones, información sobre el alias (subname) del no terminal y el código semántico asociado si es que lo tuviese.
- **Producción:** Clase encargada de la construcción de producciones. Mantiene una lista de elementos, permitiendo añadir nuevos o eliminar los existentes.
- **SemanticProduction:** Clase encargada del mantenimiento del código semántico asociado a un no terminal. Consiste en salvaguardar el texto plano correspondiente.

5 - Diseño e implementación

- **Subname:** Clase encargada de salvaguardar el nombre (alias) de un no terminal.
- **GuiSintactico:** Clase encargada de la interfaz de usuario de la pestaña “sintáctico”. Contiene todo lo relacionado con la interfaz que permite el funcionamiento de la aplicación: labels, botones, etc. Mantiene las listas de terminales y no terminales.
- **ParserReader:** Clase encargada de leer el fichero parser.cup y rellenar las listas de tokens, no terminales y precedencia.
- **Precedence:** Clase encargada de mantener la precedencia de operadores.
- **TestReader:** Clase encargada de realizar los tests sobre la gramática generada.

5.2.1.5 Tests Procesadores de Lenguajes 1

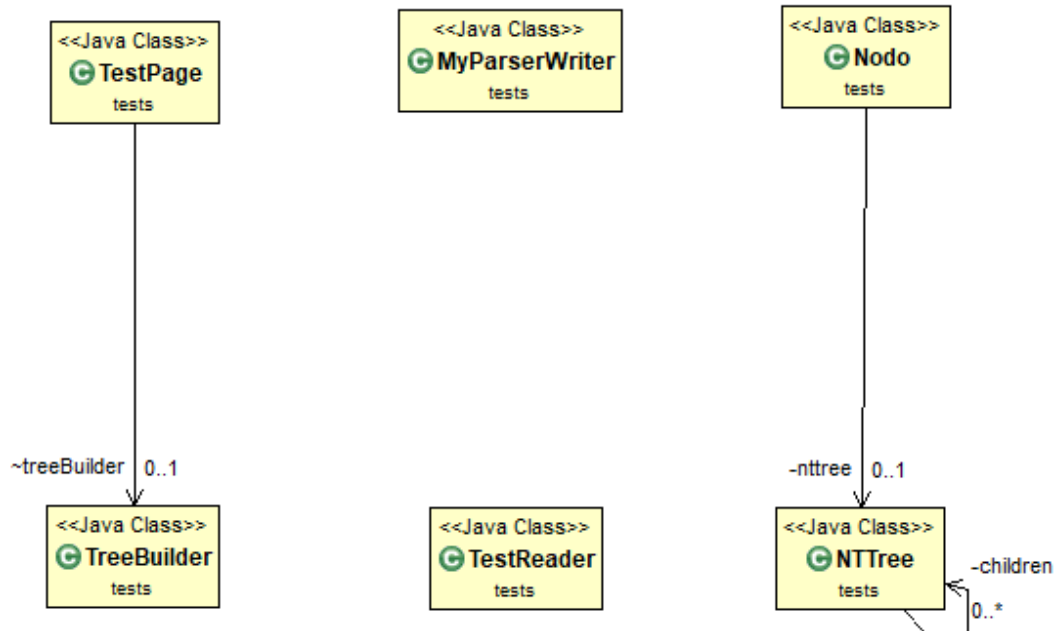


Ilustración 32 – Diagrama de clases – Tests PL1

Este paquete es el encargado de la ejecución de los tests sobre la gramática realizada. Para realizar dicha acción se genera una clase java que hace uso de CUP dentro del proyecto de Procesadores de Lenguajes del alumno, se ejecuta un código origen y se

Desarrollo de herramientas de depuración para prácticas de Procesadores del Lenguaje

captura la salida proporcionada. Si el código y la gramática son correctos se genera el árbol sintáctico correspondiente, en caso contrario se produce error.

Se detallan las clases que lo componen:

- **TestPage:** Clase encargada de la interfaz de usuario de la pestaña “Tests”. Se encarga además de localizar la carpeta donde se encuentran los tests y cargar solo aquellos con la extensión adecuada.
- **TreeBuilder:** Clase encargada de la construcción del árbol sintáctico y su exportación a fichero de texto.
- **MyParserWriter:** Clase encargada de generar la clase java auxiliar necesaria para la ejecución de CUP y la captura de la traza dentro del proyecto del alumno.
- **TestReader:** Clase encargada de leer el contenido de los tests que se van a realizar.
- **Nodo:** Esta clase representa los nodos del árbol sintáctico.
- **NTTree:** Esta clase representa el contenido de los nodos y contiene información sobre las relaciones con otros nodos (hijos, padre).

5 - Diseño e implementación

5.2.2 Diagrama de Clases Procesadores de Lenguajes 2

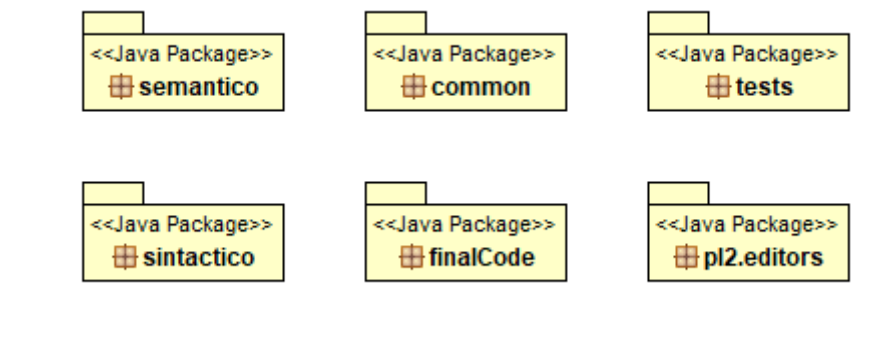


Ilustración 33 – Diagrama de clases – Paquetes PL2

Se muestran los principales paquetes del proyecto de procesadores de lenguajes 2 y se detallan a continuación. Hay que destacar que la parte de código intermedio está incluida dentro del paquete “semántico” y el tratamiento de ambas partes se realiza en el mismo paquete.

5.2.2.1 Semántico



Ilustración 34 – Diagrama de clases – Semántico

En este paquete se realiza el tratamiento del análisis semántico y código intermedio. Dicho tratamiento consiste en separar la estructura del código en texto plano para crear unas pestañas desplegables separando los diferentes no terminales, con lo que se consigue una mejor organización, y se acelera el proceso de implementación del compilador.

- **ValueExportTransferHandler:** Clase encargada de capturar los eventos “arrastrar” con los que se introduce el código mediante el arrastre de botones hacia un campo de texto.

- **GuiSemantico:** Clase encargada de la interfaz de usuario. Su funcionamiento se limita a leer el fichero parser.cup separando los no terminales unos de otros y en concreto se centra en la separación de código semántico. Esta separación se realiza en distintas pestañas desplegadas. Dentro de cada una de las pestañas se encuentran los botones arrastrables con los que junto a la clase ValueExportTransferHandler se consigue generar el código repetitivo de las prácticas de Procesadores de Lenguajes de forma ágil y rápida.

5.2.2.2 Common

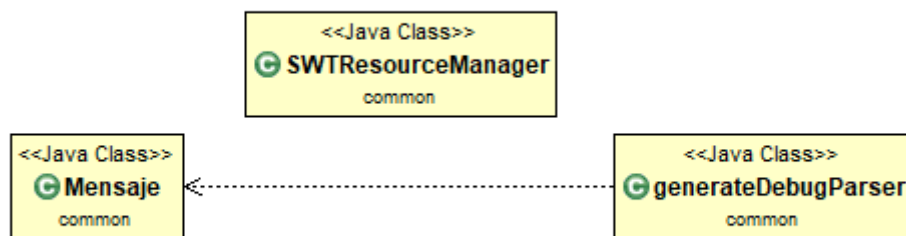


Ilustración 35 – Diagrama de clases – Common

Este paquete contiene clases útiles para todo el proyecto.

- Mensaje: Es la clase encargada de mostrar mensajes por pantalla de forma gráfica, ya sean avisos o errores.
- SWTResourceManager: Es la clase que se encarga del tratamiento de imágenes o texto, permitiendo ampliar/disminuir la fuente o aplicarle estilos a la fuente como negrita, subrayado.
- generateDebugParser: Esta clase es la encargada de crear una clase auxiliar necesaria para la ejecución de los test. Dicha clase es una clase hija del generador de analizadores de CUP, el cual emplea para su ejecución. Modifica el fichero parser.java del proyecto del alumno.

5 - Diseño e implementación

5.2.2.3 Tests Procesadores de Lenguajes 2

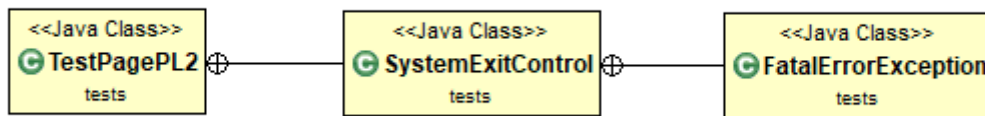


Ilustración 36 – Diagrama de clases – Tests PL2

Este paquete es el encargado de realizar los tests sobre la gramática generada. El paquete contiene un único fichero java dentro del cual se han declarado dos clases internas, SystemExitControl y FatalErrorException.

El objetivo de las clases internas es capturar un error específico lanzado por la arquitectura del Equipo Docente de la asignatura, el cual detiene la ejecución (FatalError). Esto no afecta a los alumnos en el desarrollo normal del compilador, dado que la ejecución de los tests es una ejecución java interna, pero debido a que el plugin se ejecuta con eclipse, detener la ejecución se traduce a detener eclipse en este caso, lo que produce el cierre del programa IDE. Se ha visto la necesidad de capturar este fenómeno, al menos mientras se ejecuten los tests.

- **TestPagePL2:** Esta clase es la encargada de la interfaz de usuario. Además de la ocultación del error de cierre de sistema muestra una interfaz con los tests que se encuentran en la carpeta específica del proyecto, el contenido y la ejecución paso a paso de los mismos, así como las tablas de símbolos y tipos. Para su correcto funcionamiento se genera una clase auxiliar que hace uso de CUP y modifica parser.java.

5.2.2.4 Sintáctico PL2

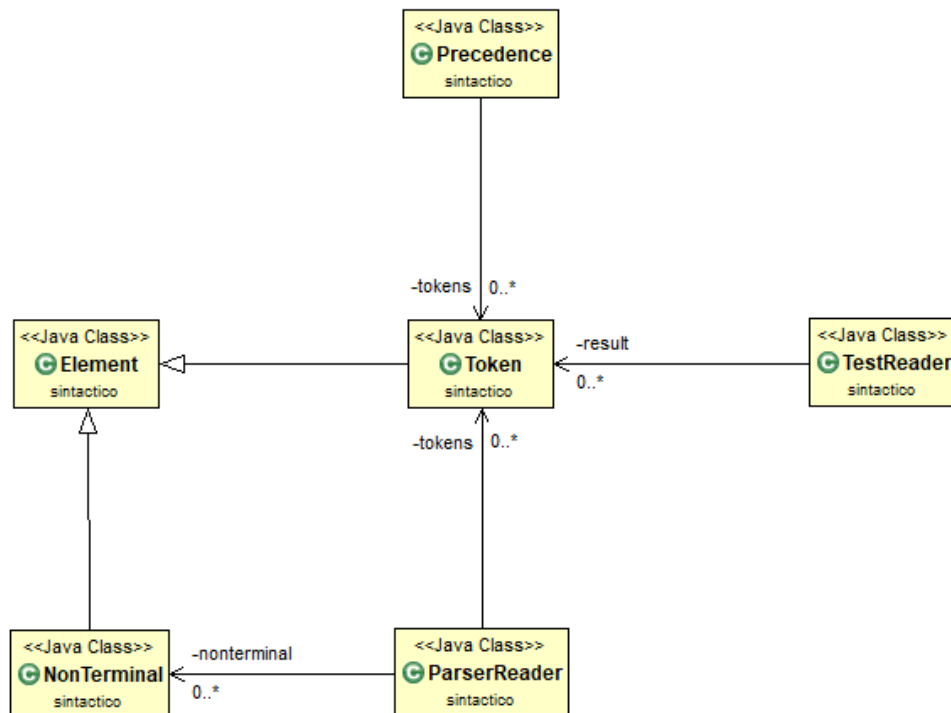


Ilustración 37 – Diagrama de clases – Sintáctico PL2

Este paquete se encarga de la lectura del parser.cup y la separación de los elementos que contiene. Es muy similar al utilizado en la primera herramienta (PL1), con la diferencia de que no posee interfaz gráfica. ComprobarErrores: Clase encargada de comprobar los posibles errores en la gramática. Se centra en la comprobación de duplicados entre los no terminales y la comprobación de que todos los no terminales utilizados en las producciones hayan sido declarados.

- **Element:** Clase padre para los elementos de la gramática, contiene el nombre del elemento, así como los métodos para comprobar si un elemento es igual a otro.
- **Token:** Clase encargada de la construcción de tokens.
- **NonTerminal:** Clase encargada de la construcción de no terminales. Mantiene una lista de producciones, información sobre el alias (subname) del no terminal y el código semántico asociado si es que lo tuviese.
- **ParserReader:** Clase encargada de leer el fichero parser.cup y rellenar las listas de tokens, no terminales y precedencia.

5 - Diseño e implementación

- **Precedence:** Clase encargada de mantener la precedencia de operadores.
- **TestReader:** Clase encargada de realizar los tests sobre la gramática generada.

5.2.2.5 FinalCode

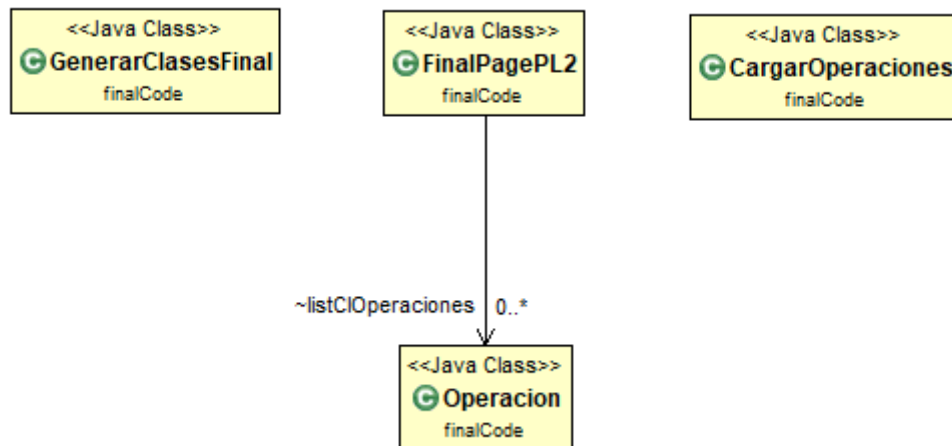


Ilustración 38 – Diagrama de clases – FinalCode

En este paquete se realiza el tratamiento de código final correspondiente a la pestaña “Código final”. Al igual que en el análisis semántico y generación de código intermedio el punto débil que va a tratarse es la estructura y organización del código. Para lograr facilitar la implementación al alumno se separan los diferentes elementos del código intermedio para poder realizar su traducción independientemente.

- **GenerarClasesFinal:** Esta clase se encarga de crear 2 clases java dentro del proyecto del alumno. La primera trata el tema del guardado de cadenas de texto y la segunda la asignación de posiciones de memoria para los símbolos (variables) del lenguaje.
- **FinalPagePL2:** Esta clase es la encargada de la interfaz de usuario. Se ocupa de la lectura y generación del fichero `ExecutionEnvironmentEns2001.java` del cual extrae las traducciones realizadas por el alumno si las hubiera.

- **Operación:** Esta clase representa cada una de las operaciones de código intermedio que nos podemos encontrar. Consiste en un par nombre-contenido para el elemento en cuestión.
- **CargarOperaciones:** Esta clase se encarga de leer el fichero parser.cup del alumno y obtener todo el código intermedio declarado con el que genera una lista de operaciones.

5.2.2.6 Paquete pl2.editors

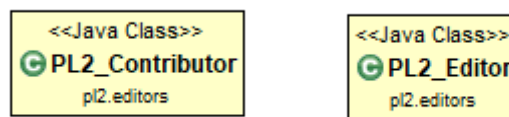


Ilustración 39 – Diagrama de clases – pl2.editors

Este paquete contiene la estructura principal de la aplicación. Se ocupa del tratamiento de las diferentes pestañas del plugin, así como el tipo de cada una de ellas, acciones a realizar al cambiar entre pestañas, diseño del logo y la ejecución.

- **PL2_Contributor:** es la clase encargada de capturar la acción que realiza el cambio entre pestañas y realizarlo.
- **PL2_Editor:** es la clase principal de este paquete. Se encarga de definir las acciones que han de ser realizadas en cada página (pestaña) del plugin. Contiene un conjunto de páginas, inicializa el plugin y mantiene funciones para el acceso y modificación de clases del proyecto.

5 - Diseño e implementación

5.3 Interfaz de usuario

A continuación, se describe la interfaz de usuario de las aplicaciones desarrolladas. Esto servirá como introducción al contenido de la interfaz de usuario que será ampliada en los anexos cuando se realice el manual de usuario.

5.3.1 Procesadores de Lenguajes 1

Tras la instalación del plugin, la ejecución será tan sencilla como acceder al fichero `scanner.flex` que se encuentra en la carpeta “`doc/specs`” de la arquitectura proporcionada para el desarrollo del compilador. A continuación, pulsamos el botón derecho del ratón sobre el fichero y abrimos con nuestro plugin (Open With > PL-1 Editor) como se muestra en la ilustración 40.

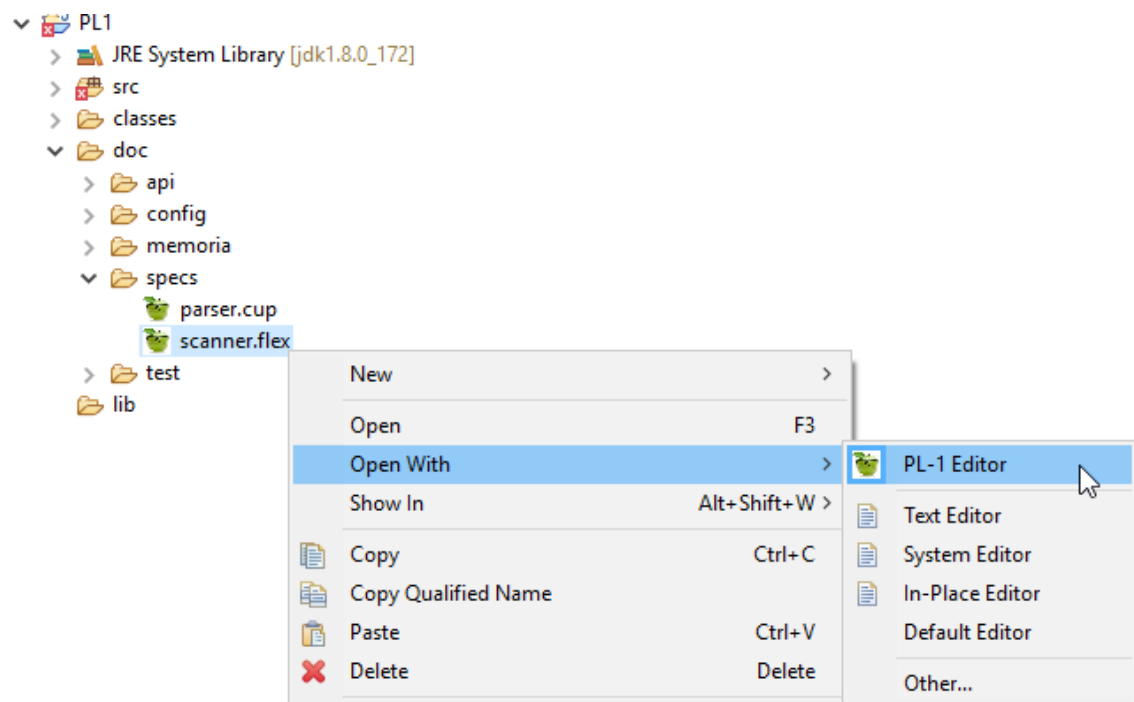


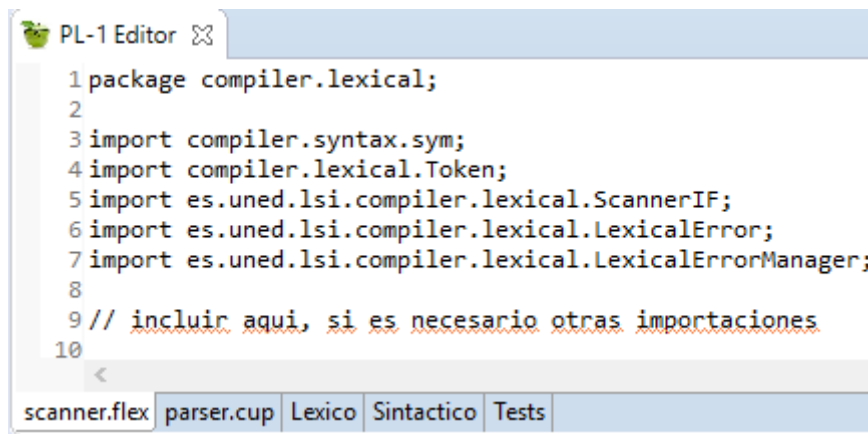
Ilustración 40 – Ejecución del plugin PL1

Una vez que hemos realizado dichas acciones se nos abrirá la pantalla principal del plugin, que siempre será la pestaña `scanner.flex` que contiene un editor de texto plano relleno con el contenido del fichero `scanner.flex`.

5.3.1.1 scanner.flex

Esta primera pestaña es un editor de texto plano dentro del cual nos encontramos el contenido del fichero scanner.flex. Cualquier modificación que se realice en el texto plano debe ser guardada para que quede reflejada en el resto de pestañas del plugin. Asimismo, cualquier cambio en el resto de pestañas debe ser guardado para que se refleje en el fichero scanner.flex y en concreto en esta pestaña.

En la ilustración 41 se muestra el contenido del plugin tras ejecutarse pudiendo ver las diferentes pestañas que contiene y en concreto la principal que es scanner.flex.



The screenshot shows a window titled "PL-1 Editor" with a tab labeled "scanner.flex". The editor contains the following Java code:

```
1 package compiler.lexical;
2
3 import compiler.syntax.sym;
4 import compiler.lexical.Token;
5 import es.uned.lsi.compiler.lexical.ScannerIF;
6 import es.uned.lsi.compiler.lexical.LexicalError;
7 import es.uned.lsi.compiler.lexical.LexicalErrorManager;
8
9 // incluir aqui, si es necesario otras importaciones
10
```

At the bottom of the window, there is a tab bar with five tabs: "scanner.flex", "parser.cup", "Lexico", "Sintactico", and "Tests". The "scanner.flex" tab is currently selected.

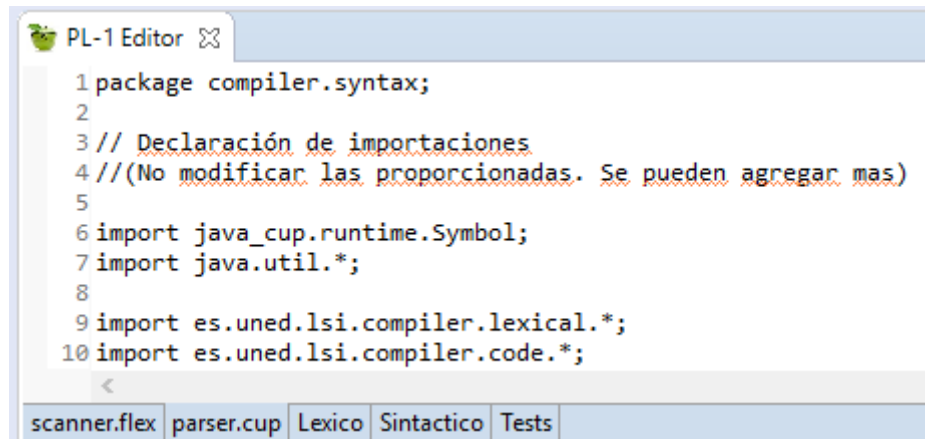
Ilustración 41 – Interfaz PL1 – scanner.flex

5.3.1.2 parser.cup

Esta segunda pestaña es un editor de texto plano dentro del cual nos encontramos el contenido del fichero parser.cup. Cualquier modificación que se realice en el texto plano debe ser guardada para que quede reflejada en el resto de pestañas del plugin. Asimismo, cualquier cambio en el resto de pestañas debe ser guardado para que se refleje en el fichero parser.cup y en concreto en esta pestaña.

En la ilustración 42 se muestra el contenido del plugin tras seleccionar la pestaña parser.cup.

5 - Diseño e implementación

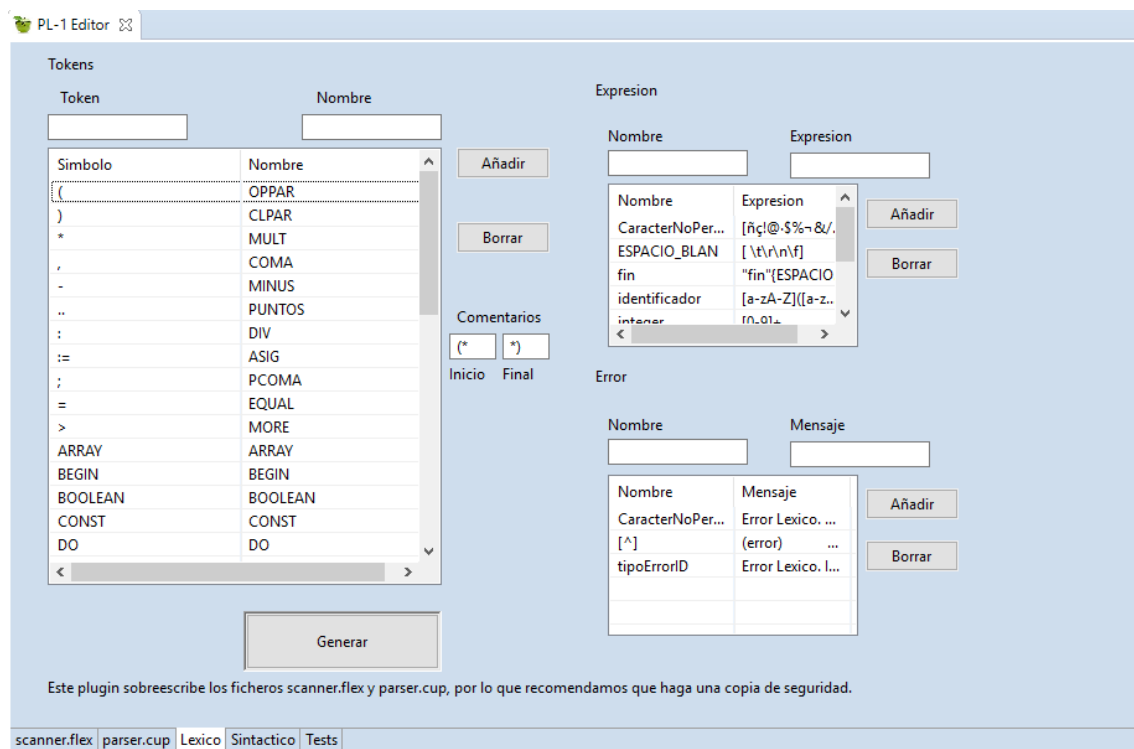


```
1 package compiler.syntax;
2
3 // Declaración de importaciones
4 //(No modificar las proporcionadas. Se pueden agregar mas)
5
6 import java_cup.runtime.Symbol;
7 import java.util.*;
8
9 import es.uned.lsi.compiler.lexical.*;
10 import es.uned.lsi.compiler.code.*;
```

Ilustración 42 – Interfaz PL1 – parser.cup

5.3.1.3 Léxico

Dentro del contenido de la tercera pestaña es donde realizaremos todo el trabajo necesario para generar un analizador léxico.



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

Ilustración 43 – Interfaz PL1 - Léxico

Se detallan las distintas partes que contiene y su funcionalidad:

- **Tokens**

The screenshot shows a window titled 'Tokens'. At the top, there are two input fields labeled 'Token' and 'Nombre'. Below these is a table with two columns: 'Símbolo' and 'Nombre'. The table contains the following entries:

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

To the right of the table are two buttons: 'Añadir' and 'Borrar'. The 'Borrar' button is disabled. At the bottom of the table, there are navigation arrows: '<' and '>'.

Ilustración 44 – Interfaz PL1 Léxico - Tokens

En esta parte se añaden los tokens de la gramática, indicando el símbolo que representan y el nombre con el que serán reconocidos dentro del analizador. Contiene una tabla con los tokens ya declarados, 2 campos de texto para la inclusión del símbolo y el nombre, botones para añadir el contenido de los campos de texto o borrar si hubiese uno seleccionado.

5 - Diseño e implementación

- **Expresión**

The screenshot shows a window titled 'Expresion'. It contains two input fields at the top, labeled 'Nombre' and 'Expresion'. Below them is a table with two columns: 'Nombre' and 'Expresion'. The table has several rows of data. To the right of the table are two buttons: 'Añadir' and 'Borrar'.

Nombre	Expresion
CaracterNoPer...	[ñç!@.\$%-&/.]
ESPACIO_BLAN	[\t\r\n\f]
fin	"fin"{ESPACIO
identificador	[a-zA-Z]([a-z...
integer	([0-9]+)

Ilustración 45 – Interfaz PL1 Léxico - Expresiones

En esta parte se añaden las expresiones de la gramática, indicando el nombre según el cual serán reconocidos dentro del analizador y la expresión (patrón) que representan. Contiene una tabla con las expresiones ya declaradas, 2 campos de texto para la inclusión del nombre y la expresión, botones para añadir el contenido de los campos de texto o borrar si hubiese uno seleccionado.

- **Error**

The screenshot shows a window titled 'Error'. It contains two input fields at the top, labeled 'Nombre' and 'Mensaje'. Below them is a table with two columns: 'Nombre' and 'Mensaje'. The table has several rows of data. To the right of the table are two buttons: 'Añadir' and 'Borrar'.

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

Ilustración 46 – Interfaz PL1 Léxico – Errores

En esta parte se añaden los errores de la gramática, indicando el nombre según el cual serán reconocidos dentro del analizador y el mensaje de error lanzado en caso de encontrarlos. Para la inclusión de los errores debe escribirse un nombre que ya haya sido declarado en la parte de expresiones, esto quiere decir que no se declara una expresión nueva para el error, sino que se utiliza una ya declarada

en la parte de expresiones. Por supuesto también es posible declarar un patrón aquí mismo. Contiene una tabla con los errores ya declarados, 2 campos de texto para la inclusión del nombre y el mensaje, botones para añadir el contenido de los campos de texto o borrar si hubiese uno seleccionado.

- **Comentarios**

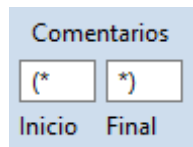


Ilustración 47 – Interfaz PL1 Léxico – Comentarios

En esta parte se añaden los comentarios del lenguaje. Un comentario es una parte del código en texto plano recibido que será ignorada por el analizador y no se procesará su contenido. En este caso concreto cualquier cosa que se encuentre delimitada por “(“ y “)” será ignorada. No hay botones para validar la inclusión dentro de alguna lista dado que solo podrán ser declarados un tipo de comentarios.

- **Guardado**

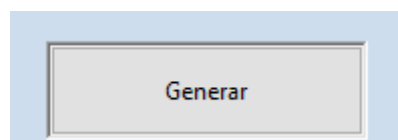


Ilustración 48 – Interfaz PL1 Léxico – Botón de guardado

Esta última parte de la pestaña Léxico es la más importante, puesto que si no se pulsa el botón generar todos los cambios producidos serán eliminados en la próxima ejecución del plugin. Generar la gramática es el equivalente a guardar el contenido en el fichero scanner.flex.

5 - Diseño e implementación

5.3.1.4 Sintáctico

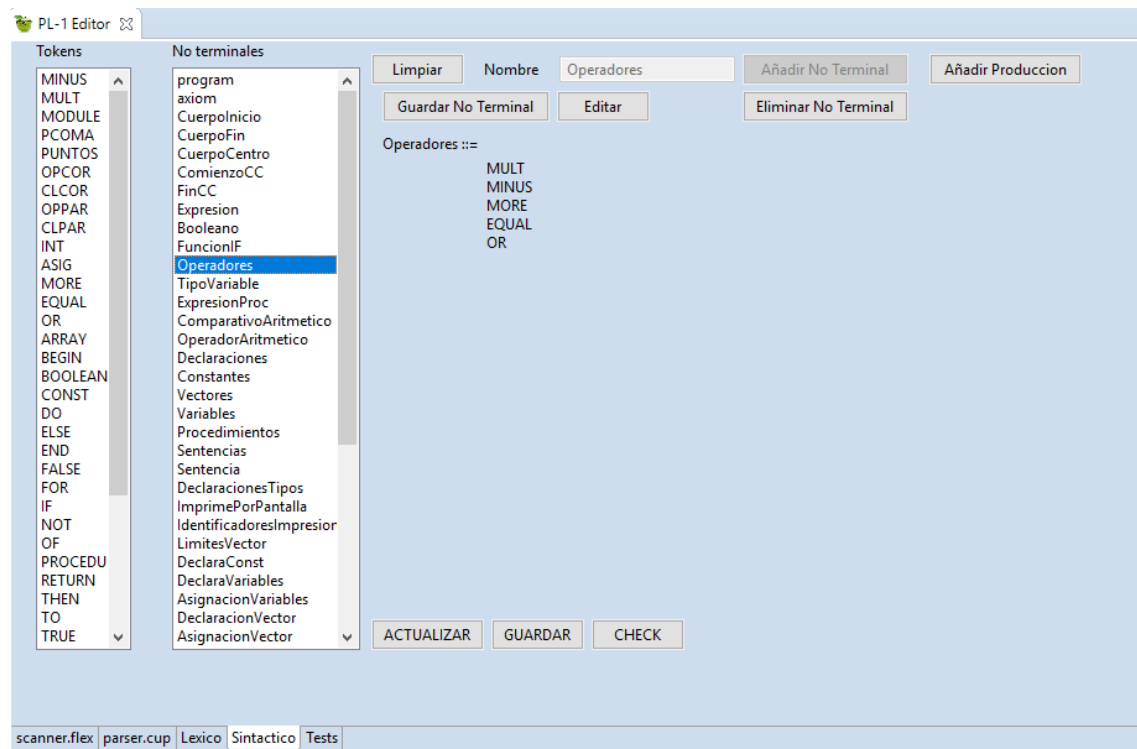


Ilustración 49 – Interfaz PL1 Sintáctico

Dentro del contenido de la cuarta pestaña es donde realizaremos todo el trabajo necesario para generar un analizador sintáctico.

Se detallan las distintas partes que contiene y su funcionalidad:

- **Tokens**

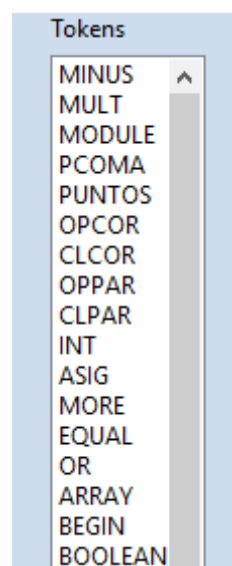


Ilustración 50 – Interfaz PL1 Sintáctico – Tokens

Esta parte contiene una lista de todos los tokens declarados en el fichero scanner.flex para que pueda verse cuales son y añadirlos a los no terminales sin tener que cambiar de pestaña o ventana.

- **No terminales**

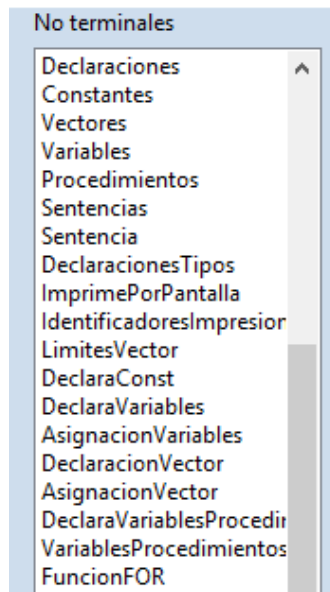


Ilustración 51 – Interfaz PL1 Sintáctico – No terminales

Esta parte contiene una lista de todos los no terminales declarados en el fichero parser.cup para que puedan añadirse a las producciones de otros no terminales y también facilitar su edición o eliminación.

- **Funciones de edición**

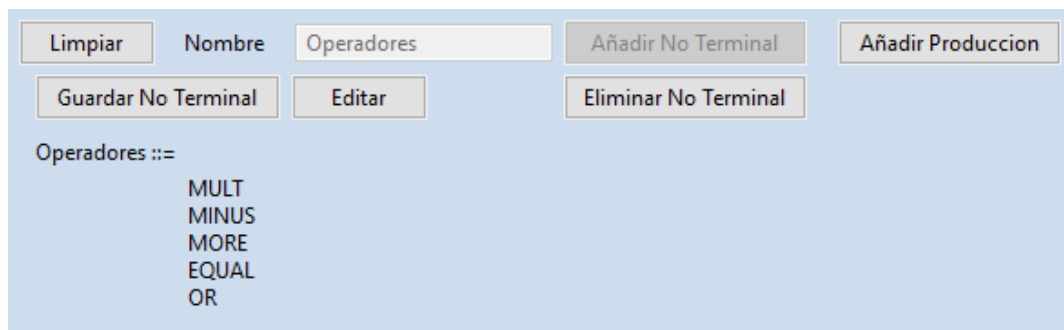


Ilustración 52 – Interfaz PL1 Sintáctico – Funciones de edición

Esta parte contiene los botones que facilitan la generación del análisis sintáctico del compilador. Nos encontramos con los siguientes:

5 - Diseño e implementación

- **Limpiar:** Elimina todo el contenido de la parte de edición
- **Nombre:** Campo de texto para incluir el nombre del no terminal antes de añadirlo. En caso de seleccionar un no terminal ya existente en la tabla de no terminales este campo se deshabilita quedando en su interior el nombre del no terminal seleccionado.
- **Añadir no terminal:** Genera un nuevo no terminal vacío y añade su contenido a la parte de edición.
- **Añadir producción:** Añade una nueva producción al no terminal seleccionado.
- **Guardar no terminal:** Guarda el no terminal generado dentro de la lista de no terminales. Este botón solo es visible cuando hay un no terminal seleccionado.
- **Editar:** Cambia la vista de la parte de edición de “solo ver” a “editar”.
- **Eliminar No Terminal:** Esta operación elimina el no terminal seleccionado.
- **Campo de edición:** En esta parte se realiza la creación de los no terminales, asignando nombres, creando producciones y sus elementos. Podemos diferenciar dos vistas dentro del campo de edición:

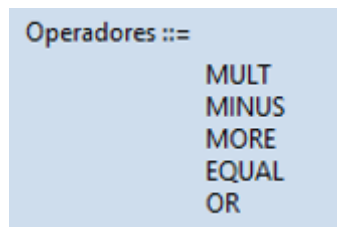


Ilustración 53 – Interfaz PL1 Sintáctico – Vista “solo ver”

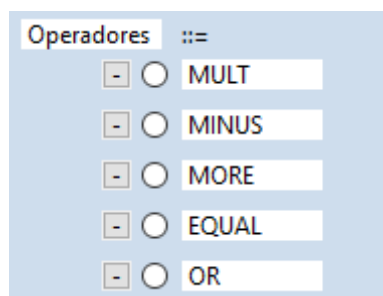


Ilustración 54 – Interfaz PL1 Sintáctico – Vista “editar”

En esta última vista (editar) encontramos dos botones que sirven para eliminar una producción o añadir elementos a la producción. Añadir un elemento se limita a añadir un campo de texto para poner el nombre del elemento, en caso de que sea solo 1 se interpretará como “épsilon”, en caso de que haya varios con contenido y otros vacíos se ignoran los campos vacíos.

- **Botones auxiliares**

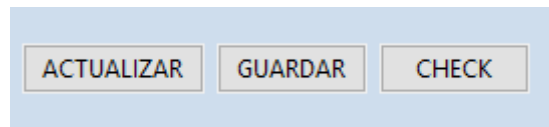


Ilustración 55 – Interfaz PL1 Sintáctico – Botones auxiliares

Esta última parte se ha separado debido a que no influye en la edición de no terminales como las otras. Tenemos 3 botones con funcionalidades totalmente diferentes:

- **Actualizar:** Refresca el contenido de las listas de tokens y no terminales, para lo cual vuelve a leer el fichero parser.cup.
- **Guardar:** Guarda el contenido de las listas de tokens y no terminales dentro del fichero parser.cup.
- **Check:** Comprueba si existen errores en la gramática generada, en concreto se centra en comprobar si existen duplicados entre los no terminales y si hay no terminales dentro de las producciones que no han sido declarados.

5 - Diseño e implementación

5.3.1.5 Tests

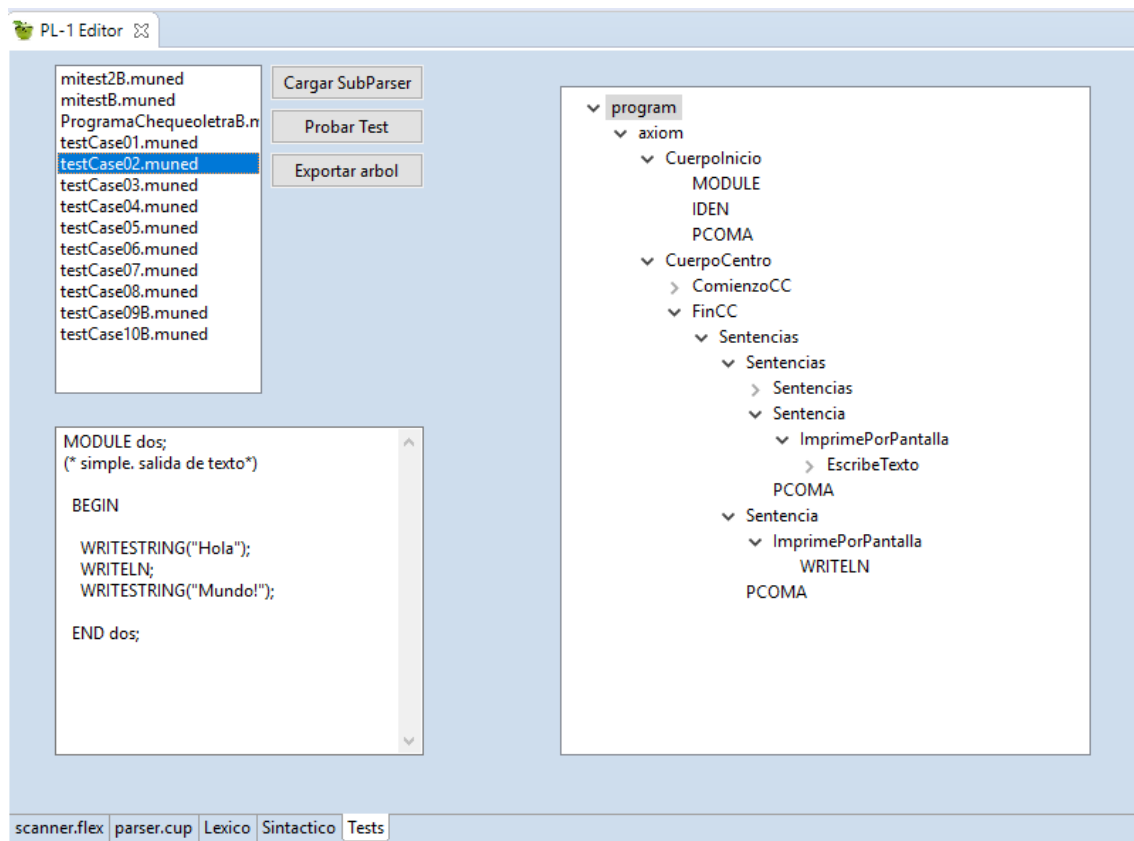


Ilustración 56 – Interfaz PL1 Tests

La última pestaña del plugin de Procesadores de lenguajes 1 trata de la realización de los tests sobre la gramática generada. En esta parte diferenciamos 4 partes: lista de tests, botones de acciones, contenido del test, representación del árbol sintáctico.

- **Lista de tests**

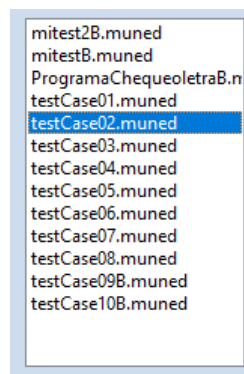


Ilustración 57 – Interfaz PL1 Tests – Lista de tests

En este apartado nos encontramos con una lista de los tests que se encuentran en la carpeta “doc/test” dentro del proyecto del alumno y tengan la extensión adecuada. Al seleccionar cualquiera de la lista se mostrará su contenido en el área de texto “contenido del test”.

- **Botones de acciones**

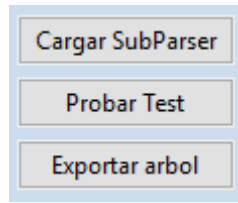


Ilustración 58 – Interfaz PL1 Tests – Botones de acciones

En esta parte nos encontramos con los 3 únicos botones que realizan alguna acción en esta pestaña del plugin.

- **Cargar SubParser:** Este botón realiza la acción de crear una clase Java llamada subparser.java dentro del paquete “compiler.syntax” dentro del proyecto del alumno. Dicha clase extiende de la clase parser de CUP y se encarga de capturar la traza realizada al ejecutar un test. Debe ejecutarse como una aplicación java (Botón derecho sobre la clase -> Run as -> Java Application) para que puedan generarse los árboles sintácticos. La ejecución generará una carpeta llamada *debug* dentro de la que podemos encontrar los resultados de los tests.

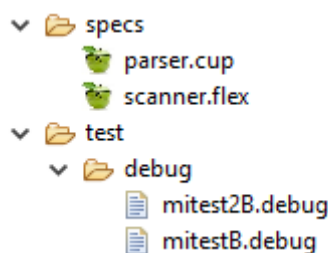


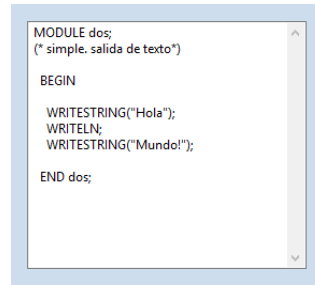
Ilustración 59 – Tests PL1 generados

- **Probar Test:** Este botón realiza la acción de ejecutar la clase subparser.java con el correspondiente test seleccionado y generar un fichero *.debug con la traza del test. A partir de dicha traza genera el árbol sintáctico.

5 - Diseño e implementación

- **Exportar árbol:** A partir del árbol sintáctico generado se genera un fichero de texto plano con su representación.

- **Contenido del test**



```
MODULE dos;  
(* simple. salida de texto*)  
  
BEGIN  
  
  WRITESTRING("Hola");  
  WRITELN;  
  WRITESTRING("Mundo!");  
  
END dos;
```

Ilustración 60 – Interfaz PL1 Tests – Contenido del test

En esta parte tenemos un área de texto que se actualiza cada vez que se selecciona uno de los tests disponibles mostrando su contenido.

- **Representación del árbol sintáctico**

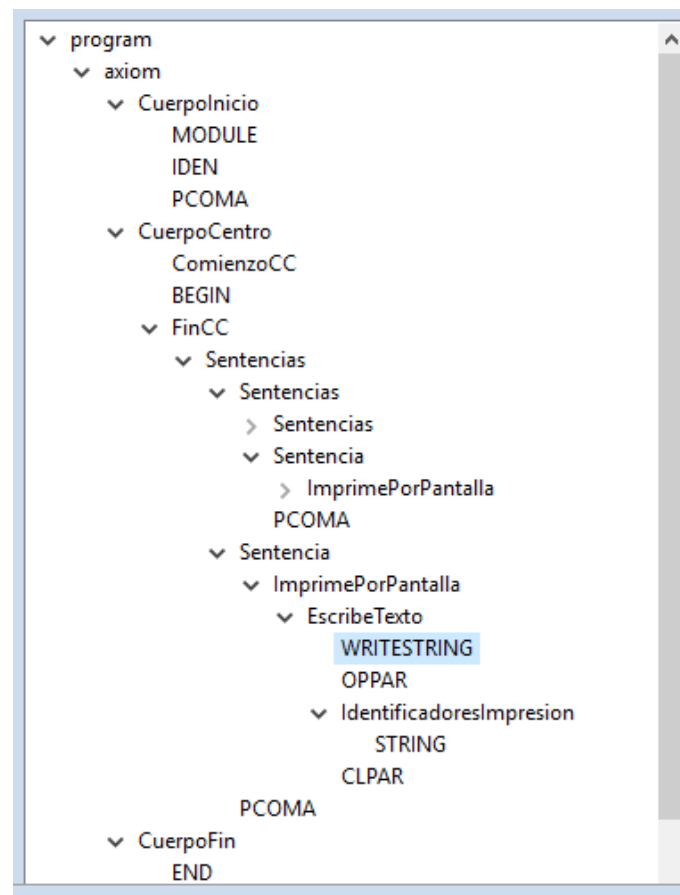


Ilustración 61 – Interfaz PL1 Tests – Árbol sintáctico

En esta parte nos encontramos con el árbol sintáctico generado, lo que no nos indica que el test se ha realizado correctamente necesariamente. El árbol generado representa cada nodo como una cadena de texto. En el caso de los no terminales dicha cadena tiene un botón a su izquierda que despliega su contenido, cosa que no tienen los terminales (tokens) que por su parte aparecen todos en mayúsculas. Es responsabilidad del alumno comprobar si el árbol generado corresponde exactamente con lo que se esperaba, en caso de que no sea así el alumno podrá comprobar en qué punto se ha dejado de rellenar el árbol y localizar mejor el error.

Por último, para la exportación del árbol en fichero de texto se ha optado por expandir el árbol completo y tabular los diferentes niveles de los nodos de la siguiente forma:

```

|program
|
|NT$0
|axiom
|
|CuerpoInicio
|
|MODULE
|IDEN
|PCOMA
|CuerpoCentro
|
|ComienzoCC
|BEGIN
|FinCC
|
|Sentencias
|
|Sentencias
|
|Sentencias
|
|Sentencia
|
|ImprimePorPantalla
|
|EscribeTexto
|
|WRITESTRING
|OPPAR
|IdentificadoresImpresion
|
|STRING
|CLPAR
|PCOMA
|Sentencia
|
|ImprimePorPantalla
|
|WRITELN
|PCOMA
|Sentencia
|
|ImprimePorPantalla
|
|

```

Ilustración 62 – Interfaz PL1 Tests – Árbol sintáctico texto plano

5 - Diseño e implementación

5.3.2 Procesadores de Lenguajes 2

Tras la instalación del plugin, la ejecución será tan sencilla como acceder al fichero `parser.cup` que se encuentra en la carpeta “`doc/specs`” de la arquitectura proporcionada para el desarrollo del compilador. A continuación, pulsamos el botón derecho del ratón sobre el fichero y abrimos con nuestro plugin (`Open With > PL2 Editor`) como se muestra en la ilustración 62.

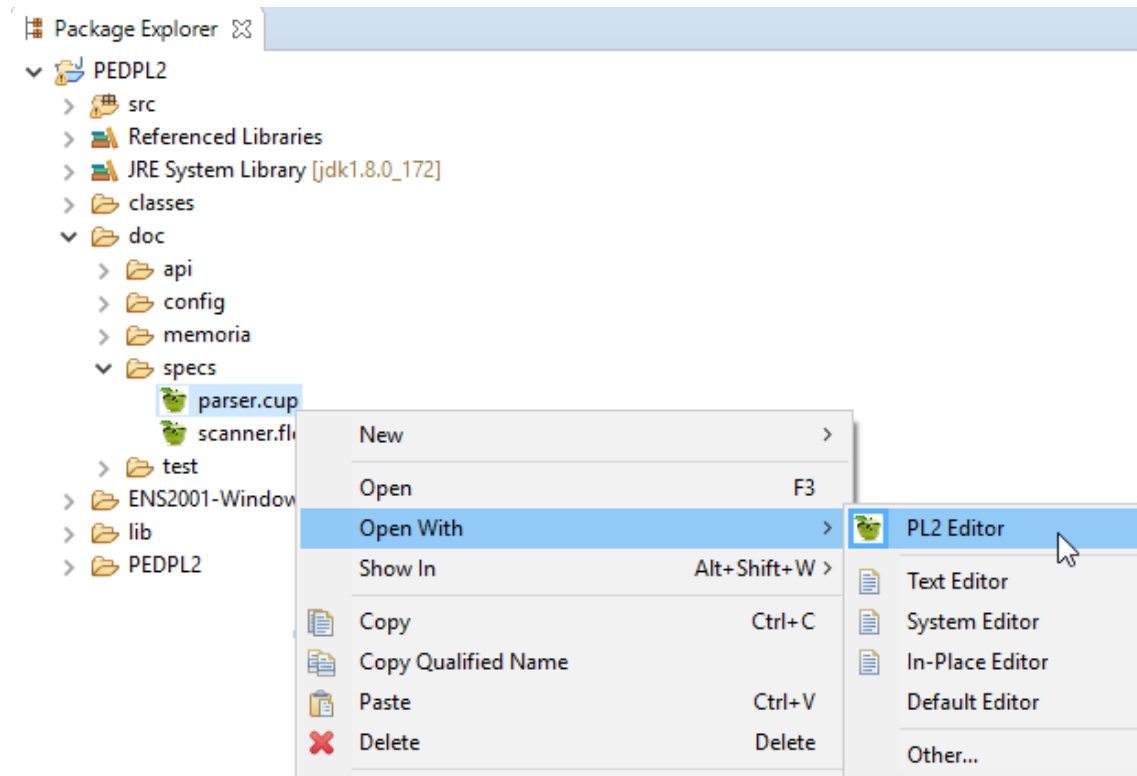


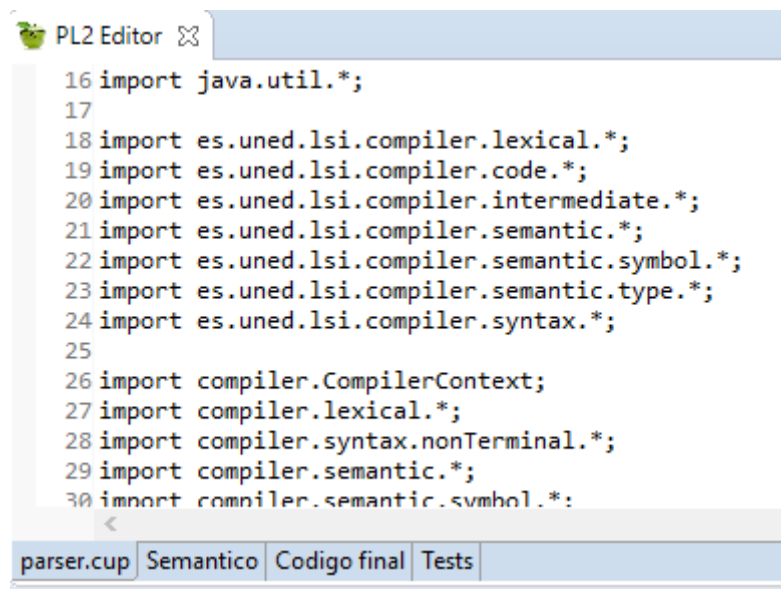
Ilustración 63 – Ejecución PL2

Una vez que hemos realizado dichas acciones se nos abrirá la pantalla principal del plugin, que siempre será la pestaña `parser.cup` que contiene un editor de texto plano relleno con el contenido del fichero `parser.cup`.

5.3.2.1 parser.cup

Esta segunda pestaña es un editor de texto plano dentro del cual nos encontramos el contenido del fichero parser.cup. Cualquier modificación que se realice en el texto plano debe ser guardada para que quede reflejada en el resto de pestañas del plugin. Asimismo, cualquier cambio en el resto de pestañas debe ser guardado para que se refleje en el fichero parser.cup y en concreto en esta pestaña.

En la ilustración 63 se muestra el contenido del plugin tras seleccionar la pestaña parser.cup.

The image shows a screenshot of the PL2 Editor application. The title bar at the top reads "PL2 Editor" with a small icon on the left and a close button on the right. The main editing area contains Java code for imports, starting with line 16: "import java.util.*;". Subsequent lines (17-30) import various classes from the "es.uned.lsi.compiler" package, including lexical, code, intermediate, semantic, semantic.symbol, semantic.type, and syntax. Line 25 imports "compiler.CompilerContext", line 27 imports "compiler.lexical.*", line 28 imports "compiler.syntax.nonTerminal.*", line 29 imports "compiler.semantic.*", and line 30 imports "compiler.semantic.symbol.*". Below the code editor, there is a tabbed interface with four tabs: "parser.cup" (which is currently selected and highlighted), "Semantico", "Codigo final", and "Tests".

```
16 import java.util.*;
17
18 import es.uned.lsi.compiler.lexical.*;
19 import es.uned.lsi.compiler.code.*;
20 import es.uned.lsi.compiler.intermediate.*;
21 import es.uned.lsi.compiler.semantic.*;
22 import es.uned.lsi.compiler.semantic.symbol.*;
23 import es.uned.lsi.compiler.semantic.type.*;
24 import es.uned.lsi.compiler.syntax.*;
25
26 import compiler.CompilerContext;
27 import compiler.lexical.*;
28 import compiler.syntax.nonTerminal.*;
29 import compiler.semantic.*;
30 import compiler.semantic.symbol.*;
```

Ilustración 64 – Interfaz PL2 – parser.cup

5.3.2.2 Semántico

En esta pestaña se realizará el tratamiento del análisis semántico y generación de código intermedio. A raíz de las investigaciones realizadas se ha considerado facilitar el desarrollo separando los diferentes no terminales en pestañas. A medida que se va realizando la práctica es fácil llegar a las 2000 líneas de código en un solo fichero de texto plano (parser.cup), lo que añade gran dificultad a la hora de relacionar un no terminal con otro, localizar un error e incluso saber qué es lo que ya está desarrollado y que es lo que falta por hacer. Por supuesto no se elimina el problema por completo, pero se facilita la localización de los no terminales al utilizar desplega- bles con el contenido de cada uno.

5 - Diseño e implementación

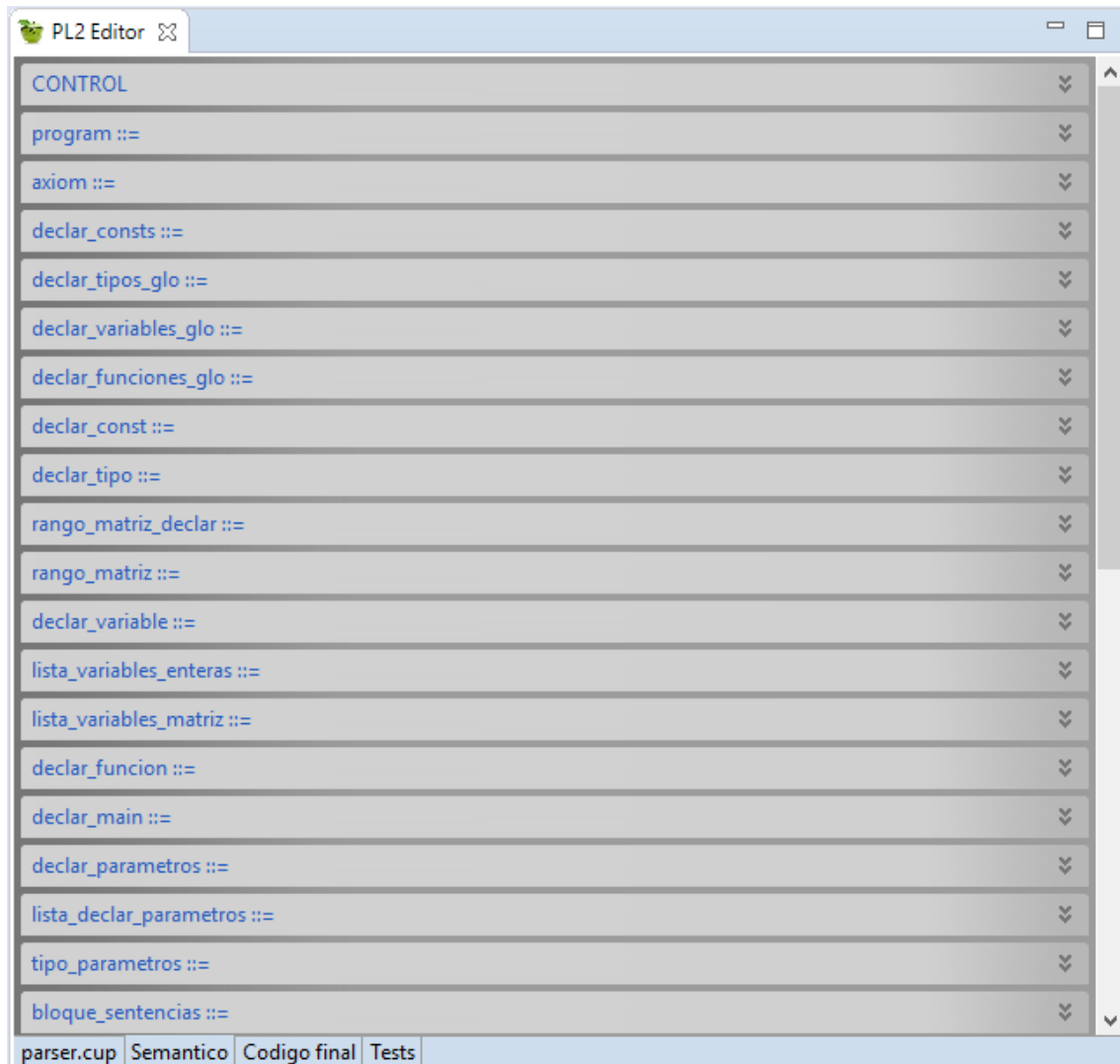


Ilustración 65 – Interfaz PL2 Semántico

Se detallan a continuación las particularidades de esta pestaña:

- **Control:**

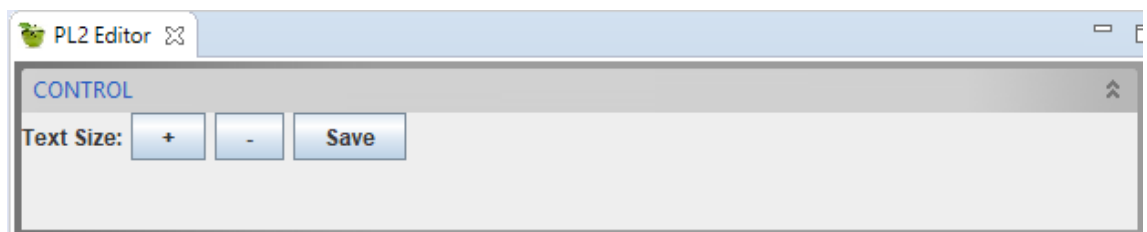


Ilustración 66 – Interfaz PL2 Semántico – Botones de control

El primer desplegable se ha reservado para tareas de control generales. En particular tenemos:

- **Aumentar tamaño del texto (+):** aumenta la fuente de todas las áreas de texto pertenecientes a la pestaña Semántico.
- **Disminuir tamaño del texto (-):** disminuye la fuente de todas las áreas de texto pertenecientes a la pestaña Semántico.
- **Guardar contenido (Save):** guarda el contenido de los desplegables dentro del fichero parser.cup.

- **Otros desplegables:**

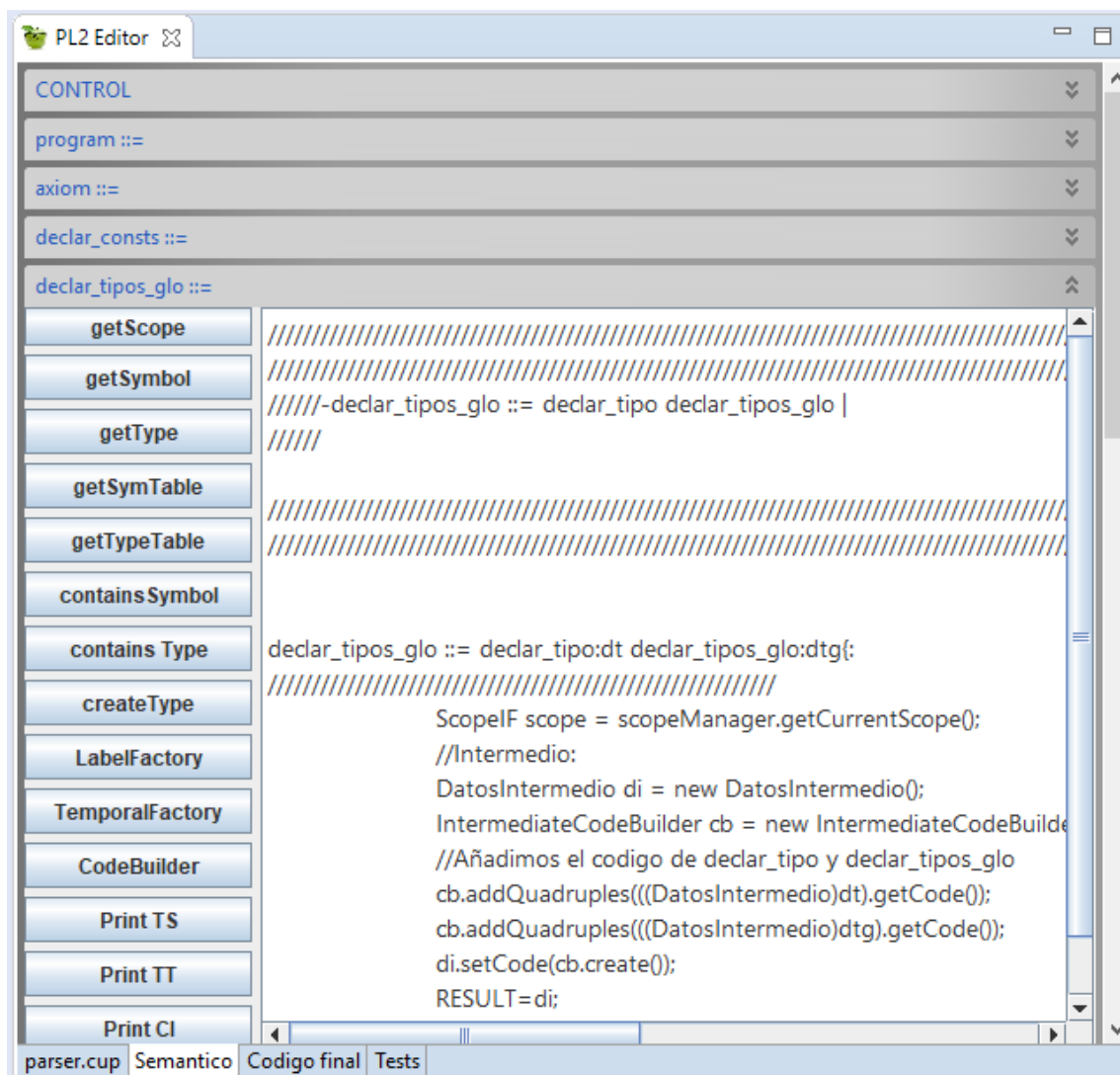


Ilustración 67 – Interfaz PL2 Semántico – Desplegables

En el resto de desplegables nos iremos encontrando el código correspondiente al no terminal que tiene por título el desplegable, en este caso “declar tipos glo”.

5 - Diseño e implementación

En la parte derecha tenemos un área de texto en el que se incluye el código.

En la parte izquierda encontramos 14 botones, cuya función es añadir texto al área de texto cuando son arrastrados en él. Se detalla a continuación el texto que añade cada botón.

- **getScope:** Obtiene el ámbito actual.

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

- **getSymbol:** Obtiene un símbolo indicando su nombre. Hay que sustituir “nombre” por el nombre del símbolo buscado.

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

- **getType:** Obtiene un tipo indicando su nombre. Hay que sustituir “nombre” por el nombre del tipo buscado.

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

- **getSymTable:** Obtiene la tabla de símbolos.

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

- **getTypeTable:** Obtiene la tabla de tipos

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

- **containsSymbol:** Comprueba si existe un símbolo indicando su nombre. Hay que sustituir “nombre” por el nombre del símbolo buscado. Guarda el resultado en una variable booleana.

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

- **contains Type:** Comprueba si existe un tipo indicando su nombre. Hay que sustituir “nombre” por el nombre del tipo buscado. Guarda el resultado en una variable booleana.

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

- **createType:** Crea un nuevo tipo de tipo simple en el ámbito actual indicando su nombre. A continuación, añade el tipo nuevo a la tabla de tipos del ámbito actual. Hay que sustituir “nombre” por el nombre del tipo nuevo que se quiera añadir.

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

- **LabelFactory:** Crea una factoría (constructor) de labels (etiquetas).

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

- **TemporalFactory:** Crea una factoría (constructor) de temporales (elementos utilizados para guardar valores en el código intermedio).

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

5 - Diseño e implementación

- **CodeBuilder:** Crea un constructor de código intermedio indicándole el ámbito (scope) actual.

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

- **Print TS:** (Print “Tabla de Símbolos”) Obtiene la tabla de símbolos del ámbito actual. Crea una lista de símbolos a partir de la tabla y por último imprime dicha lista por pantalla.

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

- **Print TT:** (Print “Tabla de tipos”) Obtiene la tabla de tipos del ámbito actual. Crea una lista de tipos a partir de la tabla y por último imprime dicha lista por pantalla.

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

- **Print CI:** (Print “Código intermedio”) Obtiene el ámbito actual, el constructor de código intermedio del ámbito, extrae el código intermedio en forma de lista y genera una cadena de texto con el contenido de las cuádruplas de código intermedio. Por último, imprime la cadena de texto.

```
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);
```

Se han elegido estos botones concretos debido a su utilidad y sobre todo a que la mayoría de estos se utilizan en varias partes del código, o incluso en casi todos los no terminales como el ejemplo de obtener el ámbito actual. Se espera que con esto se ahorre un importante tiempo al alumno al no tener que repetir el mismo código una y otra vez, facilitando que sea una función automática tan simple como arrastrar un botón. Se consigue, por tanto, disminuir el tiempo de desarrollo al estructurar el texto plano en desplegables según el no terminal que se esté tratando en el momento, y acelerar el proceso de escritura de código, todo esto sin afectar a los conocimientos técnicos necesarios para el desarrollo del compilador.

5 - Diseño e implementación

5.3.2.3 Código final

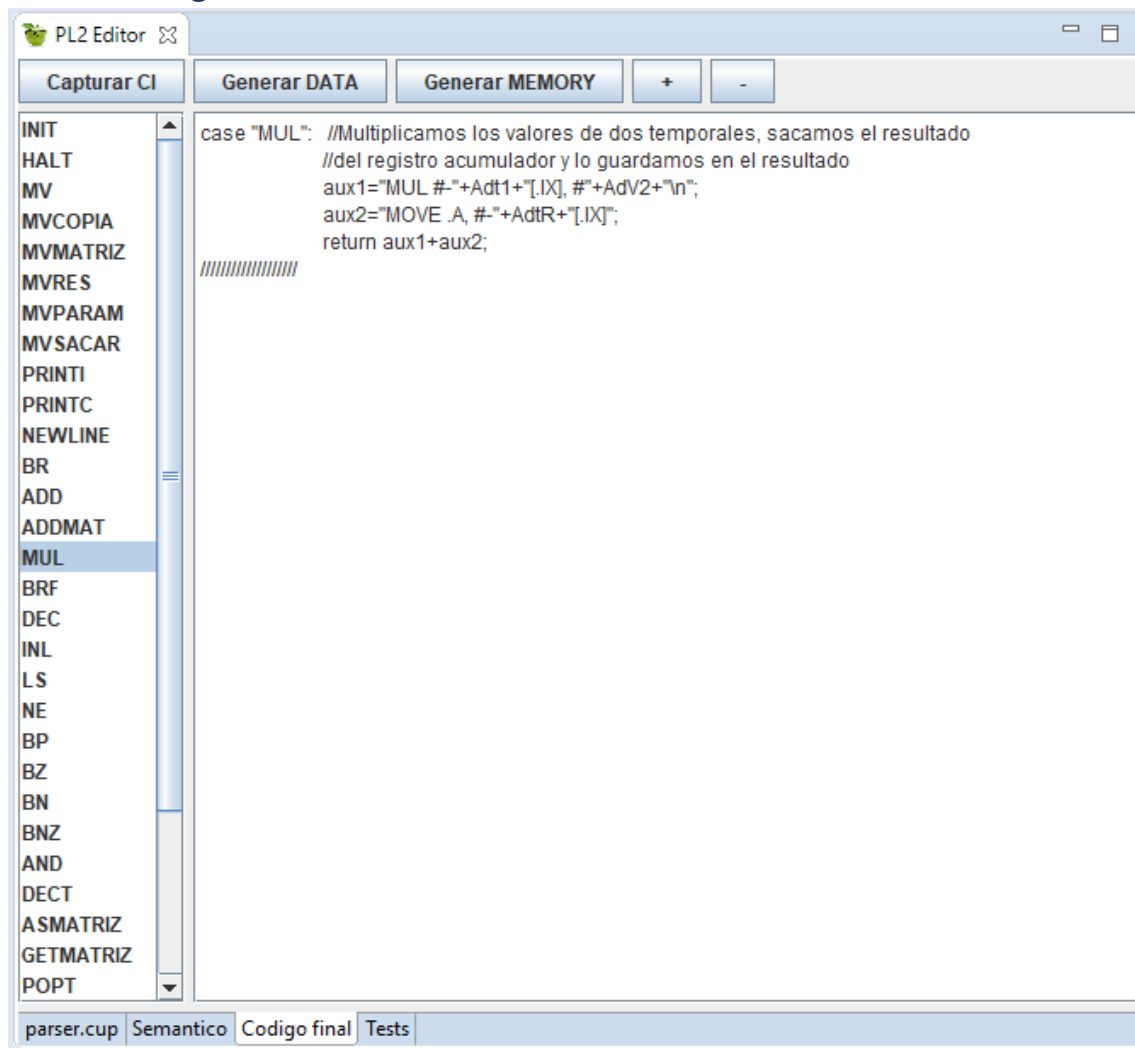


Ilustración 68 – Interfaz PL2 Código final

La tercera pestaña del plugin se ocupa del tratamiento del código final, que como ya hemos visto en las investigaciones es posiblemente la parte más compleja del desarrollo del compilador.

Se ha descartado el uso de botones para el acceso a registros o tratamiento de la memoria, puesto que están suficientemente bien explicados en el manual de Ens 2001, y realmente no aportarían una mejora considerable. En esta parte lo importante es entender cómo funciona la memoria y las instrucciones de la maquina destino y esto puede diferir mucho.

Se ha optado por modularizar el código separando las traducciones de los distintos elementos del código intermedio. Así pues, podemos diferenciar las siguientes partes en esta pestaña:

- **Botones**

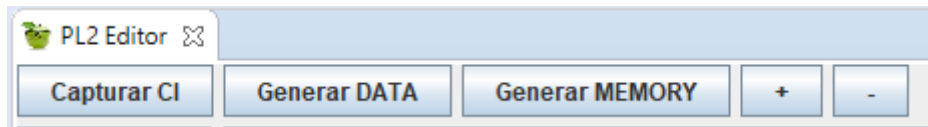


Ilustración 69 – Interfaz PL2 Código final – Botones

Las funcionalidades de los botones son las siguientes

- **Capturar CI:** Este botón realiza la acción de leer tanto el fichero `parser.cup` como `ExecutionEnvironmentEns2001.java` de las cuales recopila el código intermedio declarado y la traducción correspondiente.
- **Generar DATA:** Este botón realiza la acción de generar una clase `Data.java` dentro del proyecto del alumno, la cual ayuda a entender como guardar cadenas de texto para su posterior utilización en el código final.
- **Generar MEMORY:** Este botón realiza la acción de generar una clase `Memory.java` dentro del proyecto del alumno, la cual ayuda a entender cómo realizar la asignación de posiciones de memoria para las variables y símbolos del lenguaje desarrollado.
- **Aumentar tamaño del texto:** Realiza la acción de aumentar la fuente del texto ubicado en el área de texto.
- **Disminuir tamaño del texto:** Realiza la acción de disminuir la fuente del texto ubicado en el área de texto.

5 - Diseño e implementación

- **Lista de elementos del código intermedio**

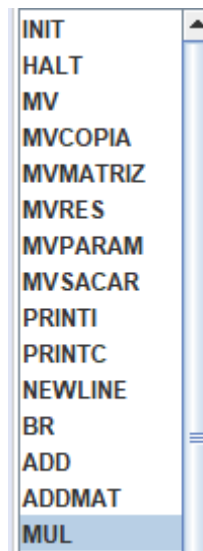


Ilustración 70 – Interfaz PL2 Código final – Lista de elementos CI

Esta parte de la pestaña consiste en una lista de elementos del código intermedio. Al hacer clic sobre cualquiera de ellos se muestra la traducción del elemento en el área de texto de las traducciones.

- **Traducción**

```
case "MUL":           //Multiplicamos los valores de dos temporales, sacamos el resultado
                      //del registro acumulador y lo guardamos en el resultado
                      aux1="MUL #-"+Adt1+"[.IX], #-"+AdV2+"\n";
                      aux2="MOVE .A, #-"+AdtR+"[.IX]";
                      return aux1+aux2;
////////////////////
```

Ilustración 71 – Interfaz PL2 Código final – Traducción

El área de texto correspondiente con las traducciones consiste en un editor de texto plano en el que puede aumentarse o disminuirse el tamaño de la fuente.

5.3.2.4 Tests Procesadores de Lenguajes 2

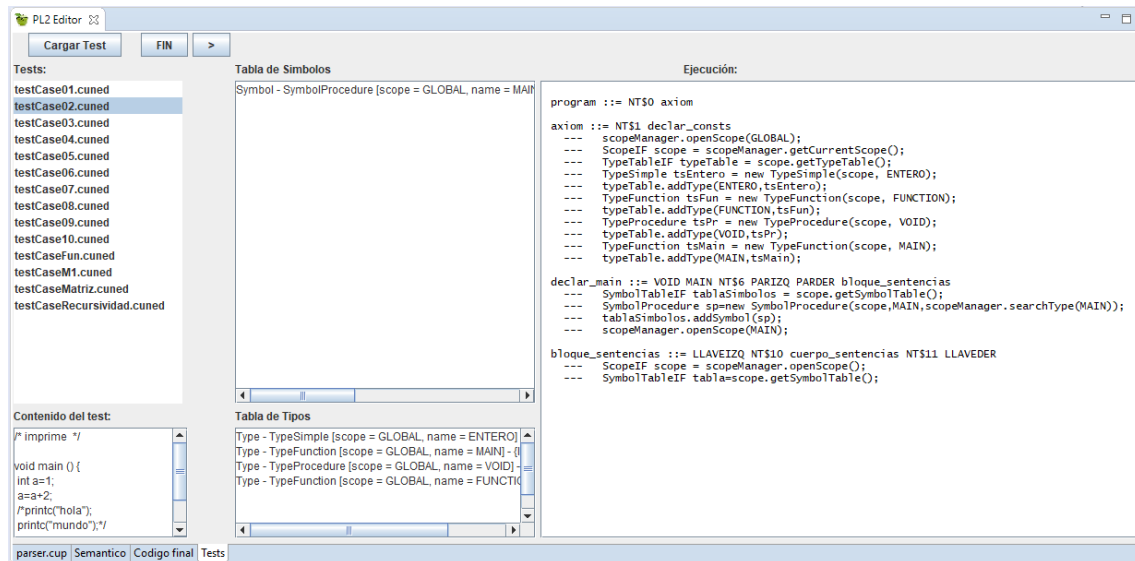


Ilustración 72 – Interfaz PL2 Tests

A partir de las investigaciones preliminares realizadas se ha concluido que una herramienta de localización de errores y ejecución paso a paso sería de gran utilidad. Tanto es así que esta pestaña ha sido uno de los principales objetivos antes incluso de empezar a plantear el proyecto.

La finalidad de esta pestaña es ayudar a los alumnos a la hora de localizar errores en el código con gran precisión. En el desarrollo del compilador fácilmente se puede llegar a 2000 líneas de código escrito en parser.cup y al producirse un error los detalles que se proporcionan no siempre ayudan a localizarlo. En muchos casos la única forma de solucionar el problema es volver a alguna copia de seguridad realizada del proyecto que funcione, ya sea de hace una hora o una semana, con la correspondiente pérdida de tiempo.

Se detallan a continuación los componentes que forman la pestaña de tests:

- **Botones**

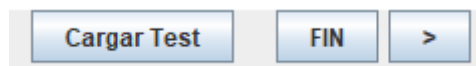


Ilustración 73 – Interfaz PL2 Tests – Botones

- **Cargar Test:** Realiza la acción de generar el archivo subparser.java en caso de que no se haya creado con anterioridad y modificar parser.java para la ejecución paso a paso.

5 - Diseño e implementación

- **FIN:** Recorre toda la ejecución del test concreto mostrándola en el área de texto de ejecución.
- **Avanzar (>):** Avanza un paso en la ejecución del test seleccionado.

- **Lista de tests**

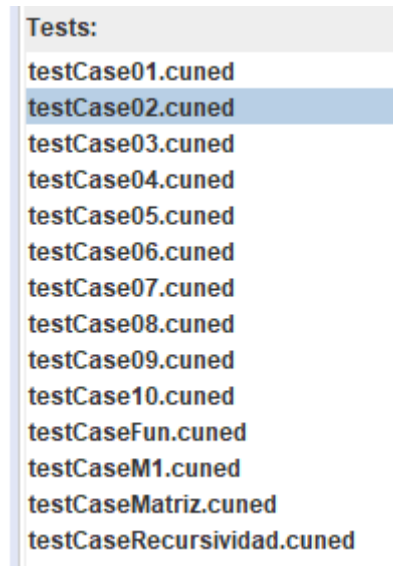


Ilustración 74 – Interfaz PL2 Tests – Lista de tests

La parte de Tests consiste en una lista ordenada de los tests encontrados en la carpeta “doc/tests” del proyecto del alumno, que coincidan con la extensión indicada. Al seleccionar un test se rellena el campo “Contenido del test” con el texto leído del fichero al que apunta.

- **Contenido del test**

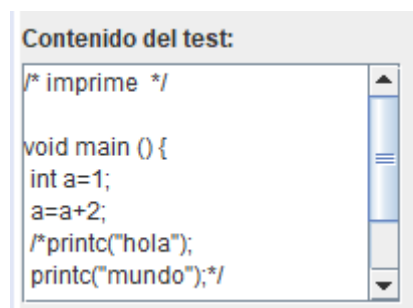
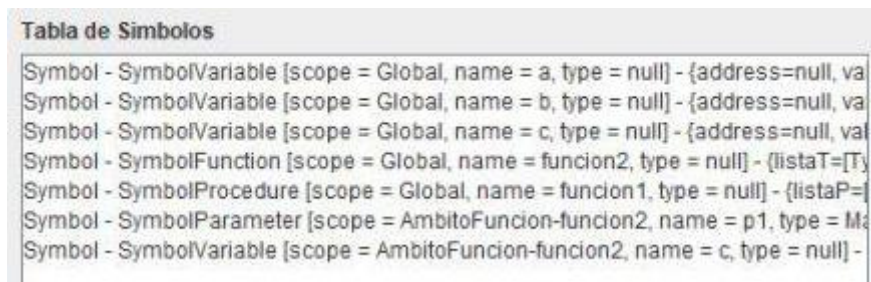


Ilustración 75 – Interfaz PL2 Tests – Contenido del test

El contenido del test es un área de texto en el que se imprime el código origen que va a ser traducido y ejecutado por el compilador.

- **Tabla de símbolos**

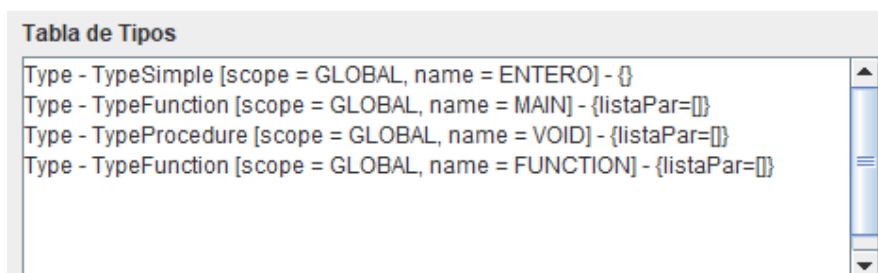


Symbol - SymbolVariable [scope = Global, name = a, type = null] - {address=null, va
Symbol - SymbolVariable [scope = Global, name = b, type = null] - {address=null, va
Symbol - SymbolVariable [scope = Global, name = c, type = null] - {address=null, va
Symbol - SymbolFunction [scope = Global, name = funcion2, type = null] - {listaT=[Ty
Symbol - SymbolProcedure [scope = Global, name = funcion1, type = null] - {listaP=
Symbol - SymbolParameter [scope = AmbitoFuncion-funcion2, name = p1, type = Ma
Symbol - SymbolVariable [scope = AmbitoFuncion-funcion2, name = c, type = null] -

Ilustración 76 – Interfaz PL2 Tests – Tabla de símbolos

Esta parte está compuesta por un área de texto dentro del cual se imprime la tabla de símbolos visibles desde el ámbito actual. Se actualiza con cada paso de la ejecución que se realice, lo que implica que si se cierra un ámbito sus símbolos dejarán de ser visibles y, por lo tanto, no aparecerán en este campo. Se ha intentado proporcionar la mayor información posible sobre cada símbolo, indicando su ámbito, nombre y tipo entre otros.

- **Tabla de tipos**



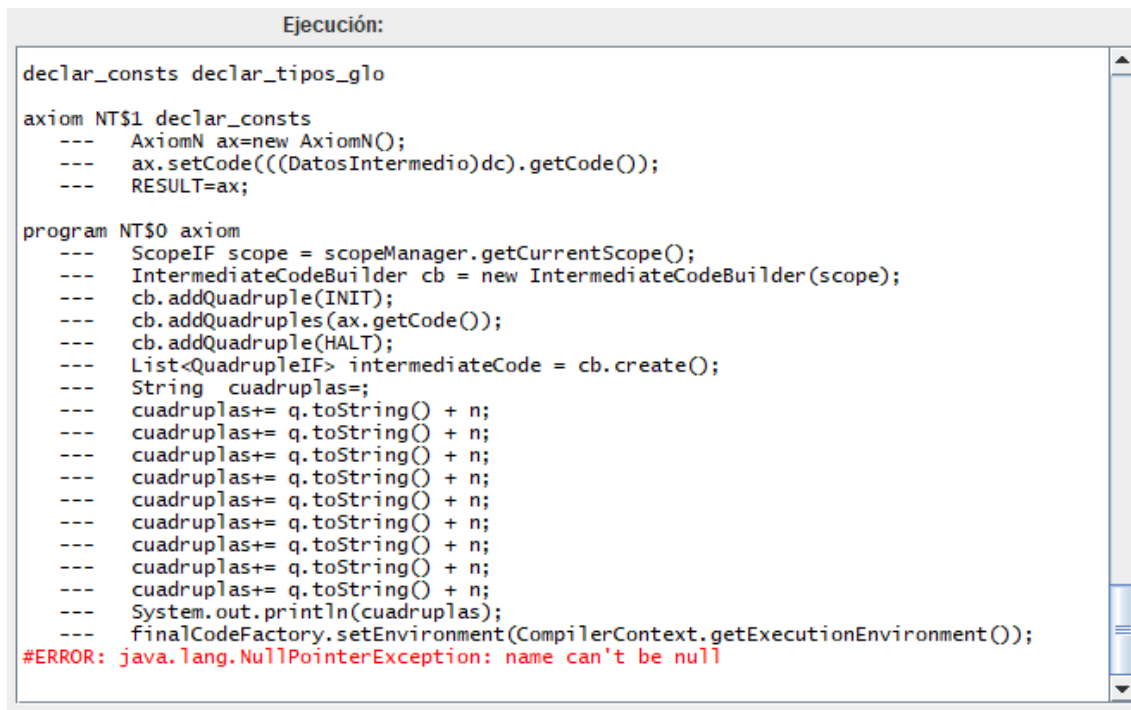
Type - TypeSimple [scope = GLOBAL, name = ENTERO] - {}
Type - TypeFunction [scope = GLOBAL, name = MAIN] - {listaPar={}}
Type - TypeProcedure [scope = GLOBAL, name = VOID] - {listaPar={}}
Type - TypeFunction [scope = GLOBAL, name = FUNCTION] - {listaPar={}}

Ilustración 77 – Interfaz PL2 Tests – Tabla de tipos

Esta parte está compuesta por un área de texto dentro del cual se imprime la tabla de tipos visibles desde el ámbito actual. Se actualiza con cada paso de la ejecución que se realice, lo que implica que si se cierra un ámbito sus tipos dejarán de ser visibles y, por lo tanto, no aparecerán en este campo. Se indican además del tipo (simple, función, procedure) el ámbito y nombre del tipo. La mayoría de los tipos se encontrarán en el ámbito global, no obstante, otros como matrices o registros pueden encontrarse en cualquier lugar.

5 - Diseño e implementación

- Ejecución del test



```

Ejecución:

declar_consts declar_tipos_glo

axiom NT$1 declar_consts
---  AxiomN ax=new AxiomN();
---  ax.setCode(((DatosIntermedio)dc).getCode());
---  RESULT=ax;

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();
---  String cuadruplas=;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  cuadruplas+= q.toString() + n;
---  System.out.println(cuadruplas);
---  finalCodeFactory.setEnvironment(CompilerContext.getExecutionEnvironment());
#ERROR: java.lang.NullPointerException: name can't be null

```

Ilustración 78 – Interfaz PL2 Tests – Ejecución del test

La funcionalidad más demandada por los alumnos es poder localizar los errores que se producen al generar el código y la ejecución paso a paso. En el área de texto correspondiente a la ejecución se realiza la traza correspondiente al test elegido. Para ello, haciendo uso del botón de avance, se imprime línea a línea las sentencias ejecutadas del código generado por el alumno.

Pegado a la izquierda encontraremos la producción que se ejecuta, y las líneas precedidas por tres guiones (---) se corresponden con la sentencia ejecutada en cada momento. Por último, en color rojo y precedido por “#ERROR” se imprimen los errores producidos, lo que detiene la ejecución del test actual. Al aparecer un error puede localizarse fácilmente al conocer la producción en la que ocurre y la línea inmediatamente anterior al error.

Además de la localización de errores, la ejecución paso a paso proporciona el beneficio de poder observar cómo avanza el código, como se mueve de un no terminal a otro y observar además el contenido de las tablas de símbolos y tipos. Se consigue una mejor comprensión sobre el compilador, lo que ayuda a reforzar conocimientos teóricos al enseñar de forma visual el funcionamiento interno.

6 Implementación

6 - Implementación

En este capítulo se describe la fase de implementación del sistema, enumerando y justificando las decisiones tomadas y explicando las dificultades encontradas.

6.1 Entorno de desarrollo

En el capítulo de Análisis del Sistema se han especificado los requisitos mínimos para la ejecución de los plugins. Cabe añadir que todo el software empleado ha sido desplegado en un ordenador personal *HP* con procesador AMD Athlon™ II X2 220 con 2 CPUs, con una velocidad de reloj de 2,8GHz y 4 GB de RAM. El sistema operativo empleado ha sido Windows 10 Home 64 bits.

En cuanto al IDE utilizado se ha optado por *Eclipse SDK Oxygen.2 Release (4.7.2)* para el desarrollo y *Eclipse IDE for Java Developers Oxygen.3 Release (4.7.3)* para realizar las pruebas.

6.2 Detalles de implementación

6.2.1 Fuentes de información

Como ya se ha tratado en el capítulo 2 “Trabajos Realizados”, solo se ha encontrado una herramienta similar al objetivo del proyecto y que abarca solo una de las asignaturas a las que va dedicado.

Debido a que el trabajo encontrado no cumplía con las expectativas del alcance deseados se ha optado por no guiarse con él y hacer una herramienta original basándose en la experiencia personal y la encuesta a los alumnos.

Cabe destacar que muchos problemas encontrados se han podido resolver gracias sitios web de StackOverFlow, Wikipedia, CUP y Eclipse.

6.2.2 CUP

Se ha estudiado la documentación relacionada con CUP, tanto en la página oficial, como de fuentes externas (Wikipedia, blogs, foros). Debido al nivel de integración con CUP que se deseaba ha sido necesario descompilar el código para entender su funcionamiento interno.

6.2.3 Patrones

En cuanto a los patrones se ha estudiado la API de Pattern de Java para realizar el reconocimiento y separación de las distintas partes de los ficheros de entrada scanner.flex y parser.cup, lo cual ha requerido de muchos intentos de prueba y error hasta acertar con el patrón correcto.

6.2.4 Plugin

La información necesaria para desarrollar un plugin se ha obtenido de la página oficial de eclipse y de los ejemplos que tiene Eclipse SDK. De nuevo la información al respecto es bastante poca por lo que se han hecho pruebas con los distintos ejemplos hasta decidir cuál encaja mejor con este proyecto.

6.2.5 WindowBuilder

El uso de WindowBuilder no ha requerido aprendizaje alguno, ya que ha sido utilizado con anterioridad y se conoce bien su funcionamiento.

6.3 Detalles de implementación

6.3.1 Diseño del plugin

En principio se optó por desarrollar una aplicación independiente de eclipse, llegando a desarrollar toda la parte de análisis léxico. Tras un estudio exhaustivo sobre el desarrollo de plugins se llegó a la conclusión que se integraría mejor con el proyecto del alumno y con CUP si formase parte de Eclipse. Con el desarrollo ya hecho y comprendiendo cómo funcionan los plugin, portar la aplicación a un plugin no ha sido una tarea compleja.

6.3.2 Diseño de las interfaces

En la herramienta de Procesadores de Lenguajes 1 se ha utilizado como contenedor de la interfaz “Composite” que tiene un diseño sencillo en cuanto a formularios y es muy fácil de utilizar.

En la herramienta de Procesadores de Lenguajes 2 se ha utilizado como contenedor de la interfaz “JFrame”, cuyo uso es igualmente sencillo, pero permite más dinamismo y tratamiento de eventos.

6 - Implementación

6.3.3 Diseño de los patrones

Esta parte supone una gran dificultad, puesto que es imposible predecir todas las posibles implementaciones que pueda realizar el alumno. Se han producido complicaciones con el “escape” de algunos caracteres como el asterisco, por lo que se han realizado pruebas aparte antes de integrar los patrones en el proyecto.

Otra parte relacionada con la búsqueda de patrones se ha encontrado en la impresión paso a paso de la herramienta de Procesadores de Lenguajes 2. No se han podido imprimir todas las líneas del código correctamente debido a que tratar de imprimir sentencias compuestas por varias líneas cortaba la ejecución de esa sentencia y producía un error.

6.3.4 Tratamiento de errores

Se ha producido una dificultad inesperada en las pruebas cerrando por completo el IDE de Eclipse. Tras varias pruebas se comprobó que el problema aparecía solo al emitirse un “SemanticFatalError” desde la arquitectura proporcionada por el Equipo Docente. Dado que dicha arquitectura no forma parte de este proyecto ha sido necesario buscar una forma de capturar dicho error y no cerrar el sistema. Este problema ha supuesto un incremento pequeño pero importante en el tiempo de desarrollo.

7 Evaluación y pruebas

7 - Evaluación y pruebas

La fase de pruebas ha sido recurrente a lo largo de todo el proceso de desarrollo del proyecto. Al haber adoptado el desarrollo iterativo y estando separado el proyecto en partes funcionales independientes, después de cada iteración se ha comprobado el funcionamiento correcto del sistema antes de avanzar.

Se detallan a continuación los casos de prueba:

7.1 Casos de prueba realizados

CP-01		Añadir Token
Herramienta	Plugin PL1	
Caso de uso	CU-1	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Léxico	
Descripción	Es la operación que permite añadir tokens al lenguaje	
Secuencia prueba	1	El usuario ingresa el símbolo del token
	2	El usuario ingresa el nombre del token
	3	El usuario pulsa el botón Añadir
	4	El sistema incluye el nuevo token en la lista correspondiente y lo muestra en pantalla.
Resultado esperado	El sistema registra correctamente el token	
Resultado obtenido	El sistema registra correctamente el token	
Resultado prueba	Correcto	

Caso de prueba 01 - Añadir Token

CP-02 Borrar Token	
Herramienta	Plugin PL1
Caso de uso	CU-1
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Léxico. Hay tokens definidos
Descripción	Es la operación que permite eliminar tokens del lenguaje
Secuencia prueba	1 El usuario selecciona un token
	2 El usuario pulsa el botón Borrar
	3 El sistema elimina el token de la lista de tokens.
Resultado esperado	El sistema elimina correctamente el token
Resultado obtenido	El sistema elimina correctamente el token
Resultado prueba	Correcto

Caso de prueba 02 - Borrar Token

CP-03 Añadir expresión	
Herramienta	Plugin PL1
Caso de uso	CU-1
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Léxico
Descripción	Es la operación que permite añadir una nueva expresión
Secuencia prueba	1 El usuario ingresa el nombre de la expresión
	2 El usuario ingresa el patrón de la expresión.
	3 El usuario pulsa el botón Añadir
	4 El sistema incluye la nueva expresión en la lista correspondiente y lo muestra en pantalla.
Resultado esperado	El sistema registra correctamente la expresión
Resultado obtenido	El sistema registra correctamente la expresión
Resultado prueba	Correcto

Caso de prueba 03 - Añadir expresión

CP-04		Borrar expresión
Herramienta	Plugin PL1	
Caso de uso	CU-1	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Léxico. Hay expresiones definidas	
Descripción	Es la operación que permite eliminar expresiones del lenguaje	
Secuencia prueba	1	El usuario selecciona una expresión
	2	El usuario pulsa el botón Borrar
	3	El sistema elimina la expresión de la lista de expresiones.
Resultado esperado	El sistema elimina correctamente la expresión	
Resultado obtenido	El sistema elimina correctamente la expresión	
Resultado prueba	Correcto	

Caso de prueba 04 - Borrar expresión

CP-05		Añadir error
Herramienta	Plugin PL1	
Caso de uso	CU-1	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Léxico	
Descripción	Es la operación que permite añadir un nuevo error al lenguaje	
Secuencia prueba	1	El usuario ingresa el nombre del error
	2	El usuario ingresa el mensaje del error
	3	El usuario pulsa el botón Añadir
	4	El sistema incluye el nuevo error en la lista correspondiente y lo muestra en pantalla.
Resultado esperado	El sistema registra correctamente el error	
Resultado obtenido	El sistema registra correctamente el error	
Resultado prueba	Correcto	

Caso de prueba 05 - Añadir error

CP-06		Borrar error
Herramienta	Plugin PL1	
Caso de uso	CU-1	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Léxico. Hay errores declarados.	
Descripción	Es la operación que permite eliminar un error del lenguaje	
Secuencia prueba	1	El usuario selecciona un error
	2	El usuario pulsa el botón Borrar
	3	El sistema elimina el error de la lista de errores.
Resultado esperado	El sistema elimina correctamente el error	
Resultado obtenido	El sistema elimina correctamente el error	
Resultado prueba	Correcto	

Caso de prueba 06 - Borrar error

CP-07		Guardar contenido
Herramienta	Plugin PL1	
Caso de uso	CU-1	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Léxico	
Descripción	Es la operación que permite guardar el contenido de las tablas de forma persistente	
Secuencia prueba	1	El usuario pulsa el botón “Generar”
	2	El sistema responde con un mensaje indicando si la operación se ha realizado.
Resultado esperado	El sistema guarda el contenido	
Resultado obtenido	El sistema guarda el contenido	
Resultado prueba	Correcto	

Caso de prueba 07 - Guardar contenido

CP-08 Añadir no terminal	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico.
Descripción	Es la operación que permite añadir un nuevo no terminal al lenguaje
Secuencia prueba	1 El usuario ingresa el nombre del no terminal
	2 El usuario pulsa el botón “Añadir no terminal”
	3 El sistema muestra el botón “Guardar no terminal”
	4 El sistema muestra el nuevo no terminal en el campo de edición
Resultado esperado	El sistema muestra el nuevo no terminal
Resultado obtenido	El sistema muestra el nuevo no terminal
Resultado prueba	Correcto

Caso de prueba 08 - Añadir no terminal

CP-09 Borrar no terminal	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico. Hay no terminales declarados.
Descripción	Es la operación que permite eliminar un no terminal
Secuencia prueba	1 El usuario selecciona un no terminal
	2 El usuario pulsa el botón “Eliminar no terminal”
	3 El sistema elimina el no terminal de la lista de no terminales
Resultado esperado	El no terminal seleccionado es eliminado
Resultado obtenido	El no terminal seleccionado es eliminado
Resultado prueba	Correcto

Caso de prueba 09 - Borrar no terminal

CP-10		Guardar no terminal
Herramienta	Plugin PL1	
Caso de uso	CU-2	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Sintáctico. Hay un no terminal en la parte de edición	
Descripción	Es la operación que permite guardar el no terminal que se encuentra en la parte de edición	
Secuencia prueba	1	El usuario pulsa el botón “Guardar no terminal”
	2	El sistema limpia la parte de edición
	3	El sistema añade el no terminal a la lista de no terminales
Resultado esperado	El no terminal es añadido a la lista	
Resultado obtenido	El no terminal es añadido a la lista	
Resultado prueba	Correcto	

Caso de prueba 10 - Guardar no terminal

CP-11		Editar no terminal
Herramienta	Plugin PL1	
Caso de uso	CU-2	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Sintáctico.	
Descripción	Es la operación que permite editar un no terminal	
Secuencia prueba	1	El usuario selecciona un no terminal
	2	El usuario pulsa el botón “Editar”
	3	El sistema muestra el no terminal y sus producciones en la parte de edición
Resultado esperado	Se muestra el no terminal en la parte de edición	
Resultado obtenido	Se muestra el no terminal en la parte de edición	
Resultado prueba	Correcto	

Caso de prueba 11 - Editar no terminal

CP-12 Añadir producción	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico. Hay un no terminal en la parte de edición
Descripción	Es la operación que permite añadir una nueva producción
Secuencia prueba	1 El usuario pulsa el botón “Añadir producción”
	2 El sistema añade una fila nueva en el campo de edición donde ingresar los datos de la producción
Resultado esperado	El sistema añade una nueva fila
Resultado obtenido	El sistema añade una nueva fila
Resultado prueba	Correcto

Caso de prueba 12 - Añadir producción

CP-13 Eliminar producción	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico. Hay un no terminal con producciones en la parte de edición
Descripción	Es la operación que permite eliminar una producción
Secuencia prueba	1 El usuario pulsa el botón “-” en la fila de la producción que desea eliminar
	2 El sistema elimina la fila elegida
Resultado esperado	El sistema elimina la fila
Resultado obtenido	El sistema elimina la fila
Resultado prueba	Correcto

Caso de prueba 13 - Eliminar producción

CP-14 Añadir componentes a la producción	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico. Hay un no terminal con producciones en la parte de edición
Descripción	Es la operación que permite incluir un nuevo elemento a una producción existente
Secuencia prueba	1 El usuario pulsa el botón radio (circulo) en la fila elegida
	2 El sistema añade un campo de texto al final de la fila para incluir el nombre del nuevo elemento
Resultado esperado	El sistema añade un nuevo campo de texto
Resultado obtenido	El sistema añade un nuevo campo de texto
Resultado prueba	Correcto

Caso de prueba 14 - Añadir componentes a la producción

CP-15 Guardar contenido	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico.
Descripción	Es la operación que permite guardar el contenido de las tablas de forma persistente
Secuencia prueba	1 El usuario pulsa el botón “Guardar”
	2 El sistema responde con un mensaje indicando si la operación se ha realizado.
Resultado esperado	El sistema guarda el contenido en parser.cup
Resultado obtenido	El sistema guarda el contenido en parser.cup
Resultado prueba	Correcto

Caso de prueba 15 - Guardar contenido

CP-16 Comprobar errores	
Herramienta	Plugin PL1
Caso de uso	CU-2
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Sintáctico.
Descripción	Es la operación que permite comprobar errores en la lista de los no terminales
Secuencia prueba	1 El usuario pulsa el botón "CHECK"
	2 El sistema responde con un mensaje con el resultado de las comprobaciones
Resultado esperado	El sistema responde con el resultado
Resultado obtenido	El sistema responde con el resultado
Resultado prueba	Correcto

Caso de prueba 16 - Comprobar errores

CP-17 Cargar SubParser	
Herramienta	Plugin PL1
Caso de uso	CU-3
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Tests
Descripción	Esta operación genera la clase subparser.java
Secuencia prueba	1 El usuario pulsa el botón "Cargar SubParser"
	2 El sistema responde indicando si se ha realizado la operación
Resultado esperado	El sistema responde con un mensaje
Resultado obtenido	El sistema responde con un mensaje
Resultado prueba	Correcto

Caso de prueba 17 - Cargar SubParser

CP-18 Probar Test	
Herramienta	Plugin PL1
Caso de uso	CU-3
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Tests.
Descripción	Esta operación permite ejecutar un test
Secuencia prueba	1 El usuario selecciona uno de los test disponibles
	2 El usuario pulsa el botón “Probar Test”
	3 El sistema responde con el resultado de la operación
	4 El sistema dibuja el árbol sintáctico
Resultado esperado	El sistema dibuja el árbol
Resultado obtenido	El sistema dibuja el árbol
Resultado prueba	Correcto

Caso de prueba 18 - Probar Test

CP-19 Exportar árbol sintáctico	
Herramienta	Plugin PL1
Caso de uso	CU-3
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Tests. Hay un árbol sintáctico generado
Descripción	Esta operación exporta el árbol a fichero de texto
Secuencia prueba	1 El usuario pulsa el botón “Exportar árbol”
	2 El sistema abre un cuadro para seleccionar la ruta y nombre del fichero
	3 El usuario selecciona la ruta y nombre
	4 El sistema guarda el árbol en el fichero
Resultado esperado	El sistema genera un fichero con el árbol
Resultado obtenido	El sistema genera un fichero con el árbol
Resultado prueba	Correcto

Caso de prueba 19 - Exportar árbol sintáctico

CP-20 Aumentar el tamaño del texto	
Herramienta	Plugin PL2
Caso de uso	CU-4
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Semántico y tiene desplegado el menú de control
Descripción	Esta operación aumenta el tamaño de fuente del texto en todos los campos de texto
Secuencia prueba	1 El usuario pulsa el botón “+”
	2 El sistema aumenta el tamaño de la fuente en los campos de texto
Resultado esperado	Aumenta el tamaño del texto
Resultado obtenido	Aumenta el tamaño del texto
Resultado prueba	Correcto

Caso de prueba 20 - Aumentar el tamaño del texto

CP-21 Disminuir tamaño del texto	
Herramienta	Plugin PL2
Caso de uso	CU-4
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Semántico y tiene desplegado el menú de control
Descripción	Esta operación disminuye el tamaño de fuente del texto en todos los campos de texto
Secuencia prueba	1 El usuario pulsa el botón “-”
	2 El sistema disminuye el tamaño de la fuente en los campos de texto
Resultado esperado	Disminuye el tamaño del texto
Resultado obtenido	Disminuye el tamaño del texto
Resultado prueba	Correcto

Caso de prueba 21 - Disminuir tamaño del texto

CP-22 Introducir código mediante botones	
Herramienta	Plugin PL2
Caso de uso	CU-4
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Semántico y tiene desplegado el menú de control
Descripción	Esta operación permite introducir código al arrastrar botones a un campo de texto
Secuencia prueba	1 El usuario mantiene pulsado un botón
	2 El usuario arrastra el botón hasta el campo de texto
	3 El usuario suelta el botón
	4 El sistema introduce un texto específico para ese botón en el sitio indicado
Resultado esperado	Se añade el texto específico del botón
Resultado obtenido	Se añade el texto específico del botón
Resultado prueba	Correcto

Caso de prueba 22 - Introducir código mediante botones

CP-23 Aumentar tamaño del texto	
Herramienta	Plugin PL2
Caso de uso	CU-5
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña “Código final”.
Descripción	Esta operación aumenta el tamaño de fuente del texto el campo de texto
Secuencia prueba	1 El usuario pulsa el botón “+”
	2 El sistema aumenta el tamaño de la fuente en los campos de texto
Resultado esperado	Aumenta el tamaño del texto
Resultado obtenido	Aumenta el tamaño del texto
Resultado prueba	Correcto

Caso de prueba 23 - Aumentar tamaño del texto

7 - Evaluación y pruebas

CP-24 Disminuir tamaño del texto	
Herramienta	Plugin PL2
Caso de uso	CU-5
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña “Código final”.
Descripción	Esta operación disminuye el tamaño de fuente del texto el campo de texto
Secuencia prueba	1 El usuario pulsa el botón “-”
	2 El sistema disminuye el tamaño de la fuente en el campo de texto
Resultado esperado	Disminuye el tamaño del texto
Resultado obtenido	Disminuye el tamaño del texto
Resultado prueba	Correcto

Caso de prueba 24 - Disminuir tamaño del texto

CP-25 Capturar código intermedio	
Herramienta	Plugin PL2
Caso de uso	CU-5
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña “Código final”.
Descripción	Esta operación permite capturar el código intermedio generado por el alumno
Secuencia prueba	1 El usuario pulsa el botón “Capturar CI”
	2 El sistema realiza la lectura de los ficheros del alumno
	3 El sistema rellena una lista con el código intermedio encontrado
Resultado esperado	Se muestra el código intermedio por pantalla
Resultado obtenido	Se muestra el código intermedio por pantalla
Resultado prueba	Correcto

Caso de prueba 25 - Capturar código intermedio

CP-26 Generar DATA	
Herramienta	Plugin PL2
Caso de uso	CU-5
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña “Código final”.
Descripción	Es la operación permite generar el fichero Data.java
Secuencia prueba	1 El usuario pulsa el botón “Generar DATA”
	2 El sistema genera el fichero Data.java en la carpeta “compiler/code” del proyecto del alumno
Resultado esperado	Se genera la clase Data.java
Resultado obtenido	Se genera la clase Data.java
Resultado prueba	Correcto

Caso de prueba 26 - Generar DATA

CP-27 Generar MEMORY	
Herramienta	Plugin PL2
Caso de uso	CU-5
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña “Código final”.
Descripción	Es la operación que permite generar el fichero Memory.java
Secuencia prueba	1 El usuario pulsa el botón “Generar MEMORY”
	2 El sistema genera el fichero Memory.java en la carpeta “compiler/code” del proyecto del alumno
Resultado esperado	Se genera la clase Memory.java
Resultado obtenido	Se genera la clase Memory.java
Resultado prueba	Correcto

Caso de prueba 27 - Generar MEMORY

CP-28	Cargar Test	
Herramienta	Plugin PL2	
Caso de uso	CU-6	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Tests.	
Descripción	Es la operación que permite ejecutar un test	
Secuencia prueba	1	El usuario selecciona un test
	2	El usuario pulsa el botón Cargar Test
	3	El sistema limpia el contenido de todos los campos de texto
	4	El sistema muestra el contenido del test en el campo de texto correspondiente
Resultado esperado	Se muestra el contenido del test	
Resultado obtenido	Se muestra el contenido del test	
Resultado prueba	Correcto	

Caso de prueba 28 - Cargar Test

CP-29	FIN (Recorre toda la ejecución)	
Herramienta	Plugin PL2	
Caso de uso	CU-6	
Versión	1.0 (08/06/2018)	
Actor	Usuario	
Precondición	El usuario se encuentra en la pestaña Tests. Se ha cargado un test.	
Descripción	Es la operación que permite finalizar la ejecución de un test mostrando toda la traza	
Secuencia prueba	1	El usuario pulsa el botón "FIN"
	2	El sistema rellena los campos de texto con el contenido de los resultados del test
Resultado esperado	El test finaliza y se rellenan los campos de texto	
Resultado obtenido	El test finaliza y se rellenan los campos de texto	
Resultado prueba	Correcto	

Caso de prueba 29 - FIN (Recorre toda la ejecución)

CP-30 Ejecución paso a paso	
Herramienta	Plugin PL2
Caso de uso	CU-6
Versión	1.0 (08/06/2018)
Actor	Usuario
Precondición	El usuario se encuentra en la pestaña Tests. Se ha cargado un test.
Descripción	Es la operación que permite ejecutar el siguiente paso del test
Secuencia prueba	1 El usuario pulsa el botón ">"
	2 El sistema escribe el siguiente paso en el campo de texto de la ejecución
	3 El sistema actualiza la tabla de símbolos y la tabla de tipos
Resultado esperado	Se imprime el siguiente paso del test
Resultado obtenido	Se imprime el siguiente paso del test
Resultado prueba	Correcto

Caso de prueba 30 - Ejecución paso a paso

8 Conclusiones y trabajo futuro

8 - Conclusiones y trabajo futuro

En este capítulo se detallan las conclusiones extraídas de la realización del proyecto, indicando si se han cumplido los objetivos.

Además, se hará mención a posibles líneas de trabajo futuro que hayan quedado pendientes por realizar.

8.1 Conclusiones

Tras la realización del proyecto de fin de grado he de decir que las expectativas en cuanto a lo que esperaba conseguir se han cumplido. Los dos plugins construidos facilitan el desarrollo del compilador facilitando el proceso y ahorrando tiempo a los alumnos, lo que se traduce finalmente en un rápido aprendizaje de la asignatura y superación de la práctica.

Como limitación encontrada en el desarrollo del proyecto se debe mencionar la dificultad a la hora de simplificar y facilitar el desarrollo del código final. Siendo dependiente de la máquina destino es necesario crear una extensión por cada tipo de máquina y aun haciéndolo lo que aportaría este plugin sería mínimo.

8.2 Trabajo futuro

Se enumeran las posibles líneas de trabajo futuro sobre el presente PFG:

8.2.1 Herramienta educativa

El propósito principal de este proyecto es orientarlo a la enseñanza, en concreto a las asignaturas de Procesadores de Lenguajes de tercer curso del Grado de Ingeniería Informática de la UNED. No obstante, se considera que las herramientas desarrolladas pueden ser utilizadas por otros en el ámbito educativo con el fin de facilitar a los alumnos la comprensión sobre el funcionamiento de los compiladores.

8.2.2 Generador rápido de compiladores

El proyecto es ideal para generar compiladores en un tiempo escaso y con un esfuerzo mínimo. Podrían integrarse las librerías de CUP y JFlex y hacer un programa completo para la generación de compiladores.

8.2.3 Nuevas funcionalidades

- Exportar el contenido del compilador en formato xls
- Dibujar el árbol sintáctico con formato visual de los nodos
- Dibujar el autómata LALR producido por CUP
- Incorporar la ejecución del código final dentro del plugin

9 Glosario y lista de acrónimos

9 - Glosario y lista de acrónimos

Se muestran todos los términos, acrónimos y abreviaturas utilizados en esta memoria:

PL	Procesadores de Lenguajes
PFG	Proyecto de Fin de Grado
JRE	Java Runtime Environment
JDK	Java Development Kit
CU	Caso de uso
CP	Caso de prueba
CI	Código Intermedio
UNED	Universidad Nacional de Educación a Distancia

10 Bibliografía

10 - Bibliografía

Referencias

- [1] *CUP Eclipse plugin* [en línea],
<http://www2.cs.tum.edu/projekte/cup/eclipse.php> [Consulta 20/02/18]
- [2] Encuesta realizada para el proyecto [en línea],
<https://docs.google.com/forms/d/1pR7uN3Mf6Ita4JSY5mXVBi-1rwF6c7T4-OICNEPeZBw/edit#responses> [Consulta 25/05/18]
- [3] JFlex The Fast Scanner Generator for Java [en línea],
<http://www.jflex.de/> [Consulta 20/02/18]
- [4] CUP, LALR Parser Generator for Java [en línea],
<http://www2.cs.tum.edu/projects/cup/> [Consulta 20/02/18]
- [5] WindowBuilder | The Eclipse Foundation [en línea],
<https://www.eclipse.org/windowbuilder/> [Consulta 20/02/18]
- [6] Oracle, Software Java [en línea],
<https://www.oracle.com/es/java/index.html> [Consulta 20/02/18]
- [7] Oracle, The Reflection API [en línea],
<https://docs.oracle.com/javase/tutorial/reflect/> [Consulta 20/02/18]
- [8] Java Platform SE 7, Pattern [en línea],
<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>
[Consulta 23/02/18]
- [9] Expresiones Regulares en Java. Programación en Castellano [en línea],
https://programacion.net/articulo/expresiones_regulares_en_java_127
[Consulta 23/02/18]
- [10] Expresiones Regulares en Java – ChuWiki[en línea]
[http://chuwiki.chuidiang.org/index.php?title=Expresiones Regulares en J
ava](http://chuwiki.chuidiang.org/index.php?title=Expresiones_Regulares_en_Java) [Consulta 23/02/18]
- [11] CUP User's Manual [en línea],
<http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>
[Consulta 21/02/18]
- [12] Tutorial de JFlex y JavaCup [en línea],
<https://es.calameo.com/read/00294884382be0c0c1e5c> [Consulta
21/02/18]

[13] Jarroba, Expresiones regulares -Regex – Pattern Matching [en línea], <https://jarroba.com/busqueda-de-patron-es-expresiones-regulares/> [Consulta 25/02/18]

[14] Adictos al trabajo, Como desarrollar un plugin para Eclipse [en línea], <https://www.adictosaltrabajo.com/tutoriales/develop-eclipse-plugin/> [Consulta 25/02/18]

[15] PDE Does Plug-ins [en línea], <http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html> [Consulta 25/02/18]

[16] Help – Eclipse Platform, Plug-in Dependencies [en línea] https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Ftools%2Feditors%2Fmanifest_editor%2Fdependencies.htm [Consulta 27/02/18]

[17] Help – Eclipse Platform, Creating the plug-in project [en línea] https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Ffirstplugin_create.htm [Consulta 27/02/18]

[18] Vogella, Creating Eclipse Wizards [en línea], <http://www.vogella.com/tutorials/EclipseWizards/article.html> [Consulta 27/02/18]

[19] Eclipse, Universal Intro changes in Eclipse 3.3 [en línea], http://www.eclipse.org/eclipse/platform-ua/documents/intro_3_3_features.html [Consulta 27/02/18]

[20] Arume informática, Java Reflection [en línea], <http://www.arumeinformatica.es/blog/java-reflection-parte-1/> [Consulta 02/03/18]

[21] Oracle, Using Java Reflection [en línea], <http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html> [Consulta 02/03/18]

[22] Jarroba, Reflection en java [en línea], <https://jarroba.com/reflection-en-java/> [Consulta 02/03/18]

[23] Arquitectura Java, El concepto Java Reflection y como utilizarlo [en línea], <https://www.arquitecturajava.com/el-concepto-java-reflection/> [Consulta 02/03/18]

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:

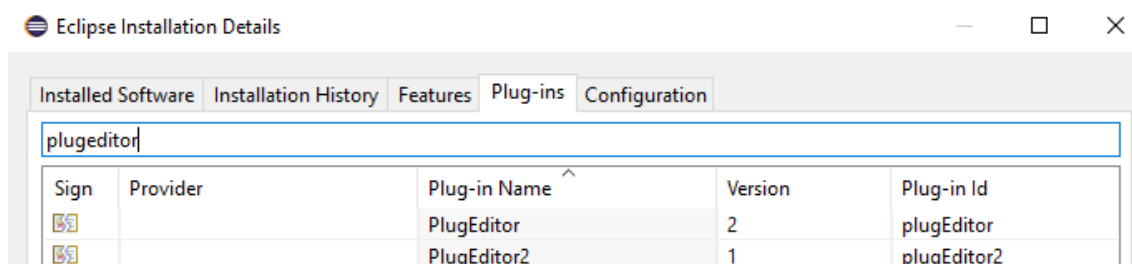


Ilustración 79 – Anexo A.2 – Proceso Instalación

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.

Tokens	No terminales
PUNTOS ^	program ^
ASIG	axiom
ARRAY	CuerpoInicio
VAR	CuerpoFin
IDEN	CuerpoCentro
ID	ComienzoCC
BEGIN	FinCC
DO	Expresion
WRITESTR	Booleano
RETURN	FuncionIF
CONST	Operadores
WRITELN	TipoVariable
OF	ExpresionProc
ELSE	ComparativoAritmetico
OPCOR	OperadorAritmetico
CLCOR	Declaraciones
TYPE	Constantes
IF	Vectores
INTEGER	Variables
INT	Procedimientos
PROCEDU	Sentencias
OR	Sentencia
WRITEINT	DeclaracionesTipos
OPPAR	ImprimePorPantalla
FOR	IdentificadoresImpresior
CLPAR	LimitesVector
TRUE	DeclaraConst
MULT	DeclaraVariables
COMA	AsignacionVariables
MINUS	DeclaracionVector
BOOLEAN v	AsignacionVector v

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

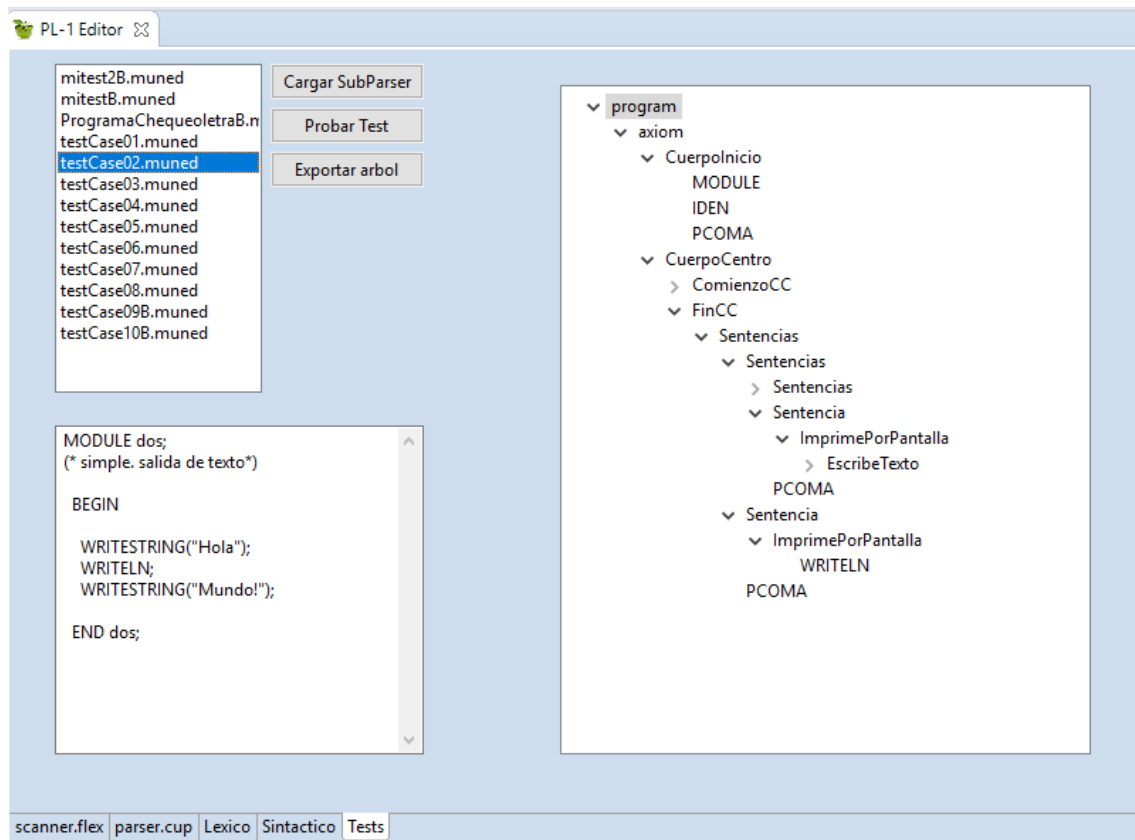


Ilustración 87 – Anexo B.1.9 – Tests PL1

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.

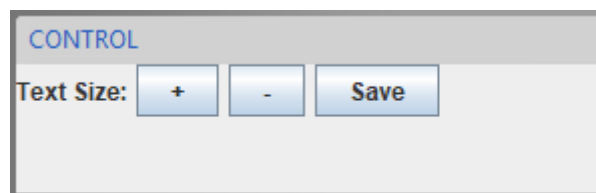


Ilustración 89 – Anexo B.2.2 – Desplegable de control PL2-Semantico

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

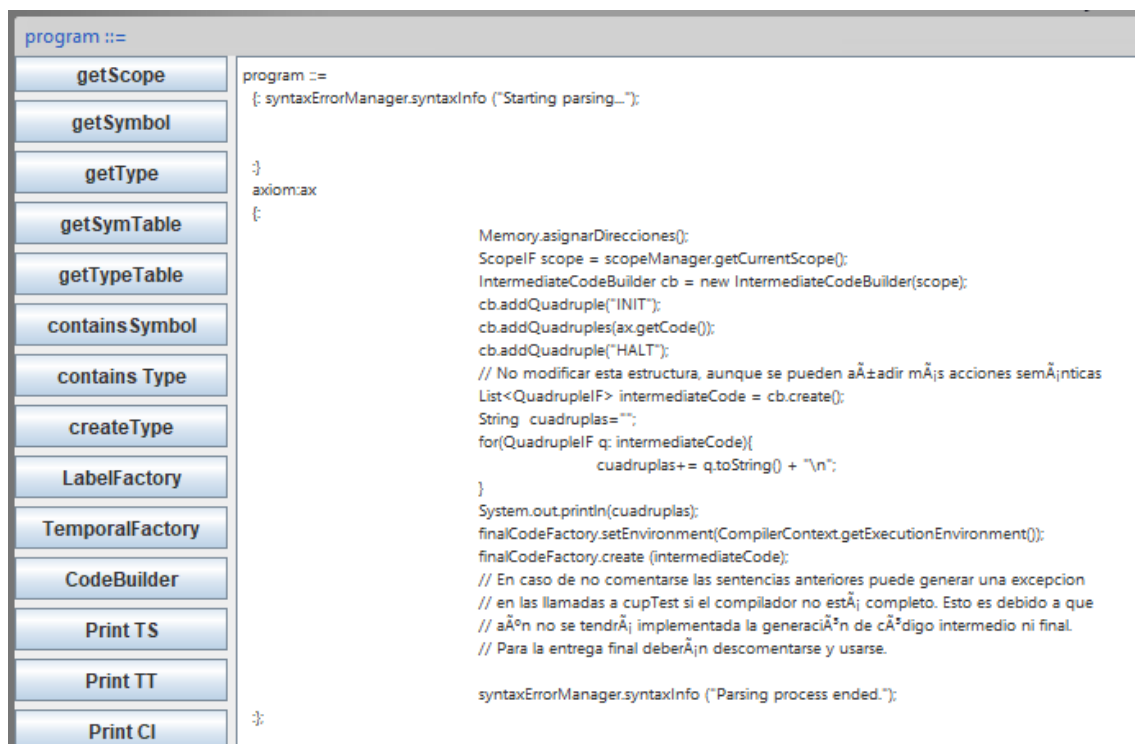


Ilustración 90 – Anexo B.2.2 – Desplegable con botones PL2-Semantico

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.

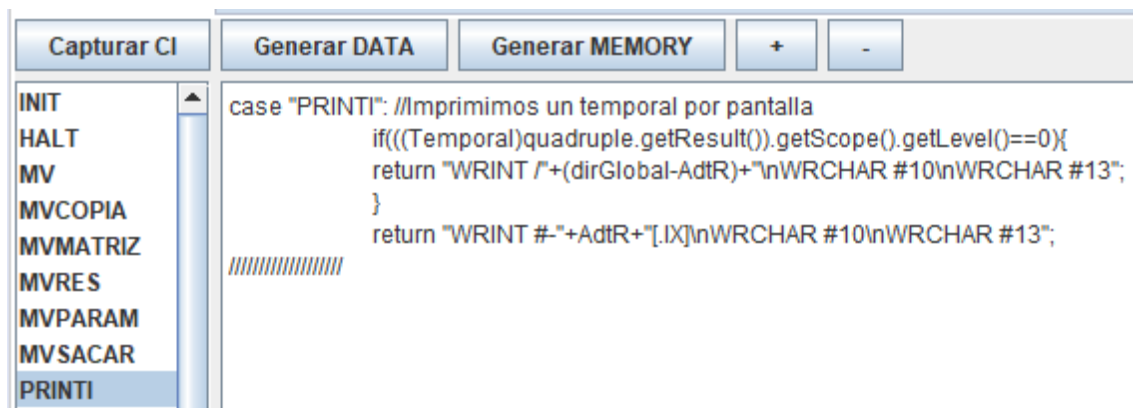


Ilustración 91 – Anexo B.2.3 – Generación de código final

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.

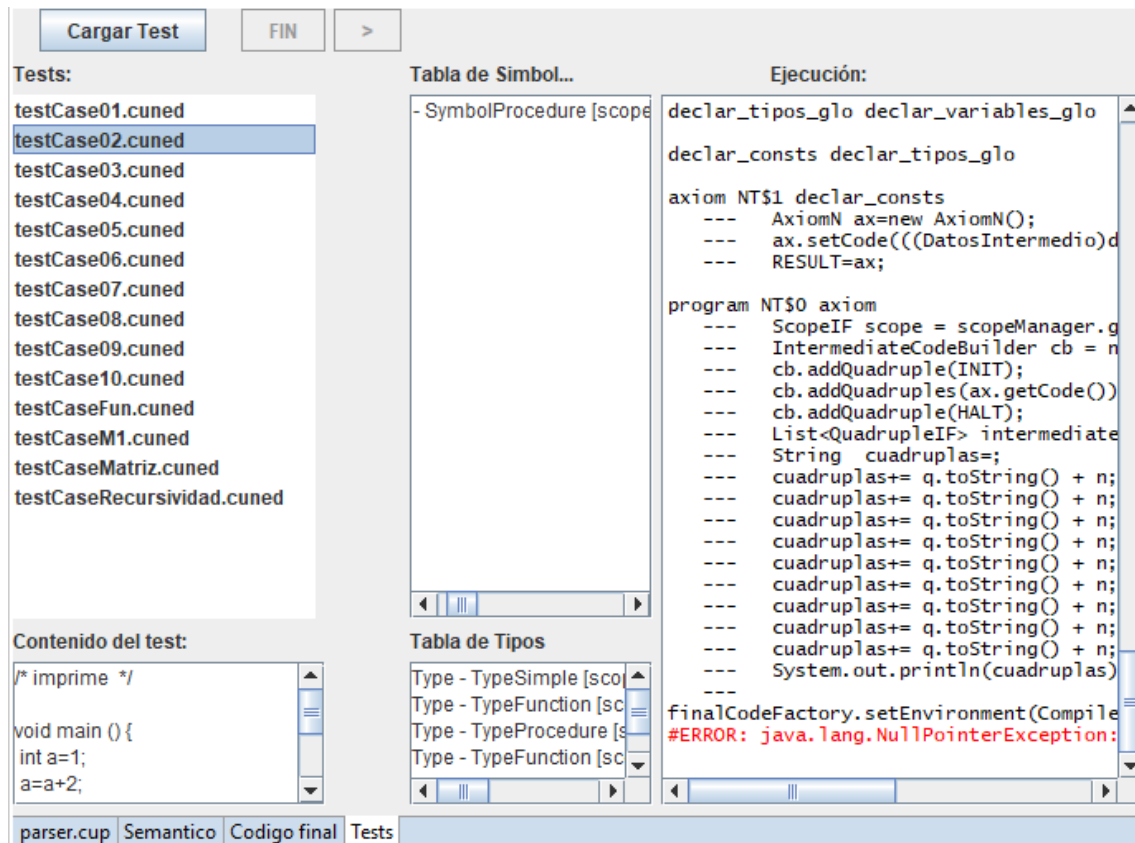
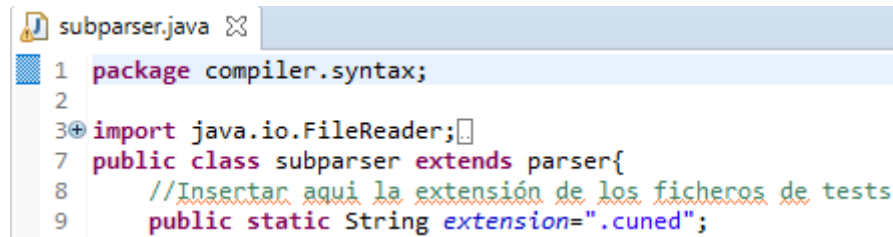


Ilustración 92 – Anexo B.2.4 – Tests en PL2

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



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

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.